A Compiled Language for Statistical Computing

Brendan McArdle

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Statistics, the University of Auckland, 2018.

Abstract

The implementation of a statistical computing system is described, consisting of a language, an optimising compiler, a virtual machine, and its run-time environment. The language is dynamically typed with lexical scope, first-class functions, and optional type declarations. The compiler operates on a high-level intermediate representation in static single assignment form, and applies several optimisations from the literature. The virtual machine is a directly threaded interpreter with specialised arithmetic instructions for unboxed scalar values. The run-time support library supplies automatic memory management of vectors, arrays and closures.

Contents

Chapter 1

Introduction

This document describes a compiler and run-time environment for a statistical computing language. In syntax and semantics it is substantially inspired by R (Ihaka and Gentleman, 1996) – lexical scope, first-class functions, ad-hoc polymorphism of arithmetic and collection subscript operators.

It includes an optimising compiler which performs type recovery, procedure integration, constant folding, and dead code elimination. The output code is interpreted by a directly threaded virtual machine.

It offers boolean, integer and double-precision floating-point scalar data types in machine-native formats, with a distinguished "missing" value for each. The compiler produces efficient code for operations on values of these types. Other types of object have reference semantics.

Unidimensional vectors and multidimensional arrays may contain values of scalar or reference type, and may have optional symbolic names attached.

1.1 Motivation

The S programming language (Becker and Chambers, 1984) was an early interactive statistical computing system. The R language (Ihaka and Gentleman, 1996), itself inspired by the later version of S (Becker et al., 1988), is a significant influence on this work. While popular and effective, some aspects of R's design preclude reliable compilation for efficient execution.

- Variables, function arguments and return values may take on values of any type, at any point in the program.
- Scalars are represented as vectors of length one.
- Call-by-value behaviour requires expensive copying, in the presence of subscript assignment.
- Reflective manipulation of the run-time environment can invalidate any static properties a compiler may have been able to deduce.

The system described in this document adopts much of R's syntax and some of its semantics. Others are included subject to minor constraints, or omitted entirely.

- Type declarations, and annotations for expressing that certain global variables have constant values.
- Values of scalar type, distinct from vectors.

- Call-by-reference for collections. Copying across function invocation may be explicitly requested by the user, if desired.
- Strictly lexical variable access. Local variables are only ever local; each may only be referred to within a single block of source code.

These changes are made in service of an optimising compiler: drawing on a selection of powerful algorithms from the computer science literature, it transforms a user's program to improve its efficiency while retaining its meaning.

A primary goal of this work is to expand the kinds of code that users can write without resorting to built-in functions for reasons of performance.

1.2 Examples

The code fragment below defines a function that returns the `n`th member of the Fibonaci sequence $1, 1, 2, 3, 5, 8, \ldots$

⟨*fib.src*⟩≡

```
const fib_rec = function(n)
    if(n <= 2) 1 else fib_rec(n - 2) + fib_rec(n - 1)
```

The syntax and semantics are similar to R, in many respects. Literals evaluate to themselves; `if` evaluates to its consequent or alternative, depending on its predicate. A call expression evaluates the actual arguments and binds their values to the formal arguments of the function (as opposed to passing the expressions, as in R). The function body is evaluated in this augmented environment, yielding the result of the call (in the absence of an explicit `return` statement).

The "`const`" declaration denotes that the global "`fib_rec`" will always hold the given value, and may not be redefined.

Following this definition, the expression `fib_rec(32)` will evaluate to `2178309`, as expected.

The function below directly calculates the `n`th term in the Fibonacci sequence (Knuth, 1997, section 1.2.8).

⟨*fib.src*⟩+≡

```
const fib.arith = function(n) {
    var sqrt5 = sqrt(5.0),
        phi = (1 + sqrt5) / 2
    as_int((phi ^ n - (1 - phi) ^ n) / sqrt5)
}
```

The local variables declared by "`var`" can have their values computed at compile time, as their initialising expressions contain only constants. The expression `fib.arith(32)` will evaluate to `2178309`, as previously.

The expression `fib.arith(1:32)` invokes the function on a vector containing the integers `1` through `32`. Vectors and arrays are passed to functions by reference. Arithmetic operations exhibit ad-hoc polymorphism, and they operate element-wise across vectors, as in R. The result will be a vector containing the first 32 terms in the sequence.

1.2.1 Linear Regression

The function below solves the linear least-squares approximation problem

$$\underset{\beta}{\operatorname{argmin}} \, ||\mathbf{y} - \mathbf{X}\beta||^2$$

with the modified Gram-Schmidt orthonormalization process (Björck, 1967). This kind of computation lies at the heart of many classical and modern statistical techniques.

⟨*regression.src*⟩≡

```
⟨lib⟩
⟨regression helpers⟩
const reg = function(array(double) X, vector(double) y,
                     double eps = 1e-7) : list
{
    var p = ncol(X), R = diag(p),
        b = dbl(p), xnorm = dbl(p), s = 0.0
    for(int j = 1; j <= p; j = j+1)
        xnorm[j] = norm(X[,j])
    for(int j = 1; j <= p; j = j+1) {
        s = norm(X[,j])
        if (s/xnorm[j] < eps)
            stop("singular X matrix")
        s = 1/s
        X[,j] = colscl(s, X[,j])
        R[,j] = colscl(s, R[,j])
        if (j < p) {
            for(int k = j+1; k <= p; k = k+1) {
                s = dotprod(X[,j], X[,k])
                X[,k] = colswp(X[,k], s, X[,j])
                R[,k] = colswp(R[,k], s, R[,j])
            }
        }
        s = dotprod(X[,j], y)
        y = colswp(y, s, X[,j])
        b = colswp(b, -s, R[,j])
    }
    vec(Q = X, residuals = y, Rinv = R, coeff = b)
}
```

The `dotprod` and `norm` functions return the dot product and Euclidean norm of their vector arguments. `colscl` scales the column `x` by `a`; `colswp` sweeps `a` multiples of the vector `x` out of `y`.

Small, immutable functions such as these can be safely expanded in-line at their call sites by the compiler. The types of their formal arguments can then be recovered from the types of the actual arguments in each function call expression.

⟨*regression helpers*⟩≡

```
const dotprod = function(x, y) sum(x * y)
const norm = function(x) sqrt(dotprod(x, x))
const colscl = function(a, x) a * x
const colswp = function(y, a, x) y - a * x
```

`nrow` and `ncol` inspect the shape vector of the array, returning count of rows and columns, respectively. The declared return type `int` allows the compiler recover the type of `p` in the function above without the user needing to declare it.

`diag` creates and returns an $n \times n$ identity matrix. As `i`, `n` and `1` are integers, the loop test and iteration will compile to specialised instructions operating on unboxed scalars. The type of `a` is also known, so the subscripted assignment to `a` will store the constant scalar directly into the addressed element.

⟨*regression helpers*⟩+≡
```
const nrow = function(array x) : int { shape(x)[1] }
const ncol = function(array x) : int { shape(x)[2] }
const diag = function(int n) : array(double) {
    let array(double) a = array(type(double), vec(n,n), 0.0)
    for(int i=1; i <= n; i = i + 1)
        a[i,i] = 1.0
    return a
}
```

A random `X` (with `n` observations of `k` variables) and `y` are created, before timing the execution of `reg`.

⟨*regression.src*⟩+≡
```
n = 10000
k = 100
X = array(type(double), vec(n, k), 0.0)
for(i=1; i<=k; i=i+1)
    X[,i] = rnorm(n) + (if(i<=k/2) 1:n else 0)
y = 1 + 0.1 * 1:n + rnorm(n,0.5)
const test = function() : double { time(function() reg(X,y))[1] }
mean(replicate(10, test))
```

With this test data, `reg` executes in an average of 0.53 seconds (over 10 runs) on test hardware equipped with an AMD FX-8350 CPU and 16GB PC3-10600 ECC DDR3 SDRAM.

This example function can be translated to R in a straightforward manner. `var`, `const` and type declarations are unnecessary; `for` has syntactic sugar for iteration over regular ranges (the former establishes necessary constraints which the compiler can use; the latter has no performance impact.)

⟨*regression.R*⟩≡
```
⟨R regression helpers⟩
reg = function(X, y, eps = 1e-7) {
    R = diag(ncol(X))
    b = rep(0, ncol(R))
    n = nrow(X)
    p = ncol(X)
    xnorm = numeric(p)
    s = 0
    for(j in 1:p)
        xnorm[j] = norm(X[,j])
    for(j in 1:p) {
        s = norm(X[,j])
        if (s/xnorm[j] < eps)
            stop("singular X matrix")
        s = 1/s
        X[,j] = colscl(s, X[,j])
        R[,j] = colscl(s, R[,j])
        if (j < p)
            for(k in (j+1):p) {
```

```
                s = dotprod(X[,j], X[,k])
                X[,k] = colswp(X[,k], s, X[,j])
                R[,k] = colswp(R[,k], s, R[,j])
            }
        s = dotprod(X[,j], y)
        y = colswp(y, s, X[,j])
        b = colswp(b, -s, R[,j])
    }
    list(Q = X, residuals = y, Rinv = R, coeff = b)
}
```

The R helper functions are identical, modulo `const`.

⟨*R regression helpers*⟩≡

```
dotprod = function(x, y) sum(x * y)
norm = function(x) sqrt(dotprod(x, x))
colscl = function(a, x) a * x
colswp = function(y, a, x) y - a * x
```

Constructing random test data is simplified by the standard library function `cbind`. `system.time` takes an expression, not a function, since it can delay its argument's evaluation until a timestamp has been recorded.

⟨*regression.R*⟩+≡

```
n = 10000
k = 100
X = cbind(replicate(k/2,rnorm(n)+1:n), replicate(k/2,rnorm(n)))
y = 1 + 1:n + .1 * rnorm(n)
mean(replicate(10, system.time(reg(X, y))[["elapsed"]]))
```

On test hardware, R version 3.5.3 takes an average of 1.36 seconds (over 10 runs) to execute `reg` on the same test data, showing this language is performant when operating over collections in a familiar style of program.

1.2.2 Closest Points

Python (van Rossum, 1995) is a general-purpose programming language which is, like the language described here, evaluated by interpreting bytecode compiled from source. When extended with the NumPy library (van der Walt et al., 2011), it is increasingly popular for numerical computation.

The next example is a variation of the nearest neighbour problem. For purposes of comparison, it will be presented in our language, R, and Python 3.7.3, respectively.

The `eucl.dist` function calculates the Euclidean distance

$$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

between the `k`-dimensional points `x` and `y`.

⟨*closest-points.src*⟩≡

```
⟨lib⟩
const eucl.dist = function(x, y) : double
    sqrt(sum(square(x - y)))
```

⟨*closest-points.R*⟩≡

```
eucl.dist = function(x, y)
    sqrt(sum((x - y) ^ 2))
```

⟨*closest-points.py*⟩≡
```
⟨imports⟩
def eucl_dist(x, y):
    return np.sqrt(np.sum(np.power(x - y, 2)))
```

Given a vector `xs` of such points, the index (in another vector `ys`) of the closest point to each can be found by computing the distance of each `x` to every `y` and recording the index of the latter where the minimum is found. This simple algorithm takes $O(n^2)$ time. [1]

`closest.map.list` is a straightforward translation of this algorithm. A `map` over the `xs` forms an intermediate vector of distances for each `x` with a nested `map`, from which the index of the minimum is returned by `which.min`.

(The brevity of the Python translation is due to its list comprehension operator, which is syntactic sugar for expressing just such an iteration.)

⟨*closest-points.src*⟩+≡
```
const closest.map.list = function(vector(numeric) xs,
                                  vector(numeric) ys) : integer
    map(xs, function(x) : int
              which.min(map(ys, function(y) : double
                                  eucl.dist(x, y))))
```

⟨*closest-points.R*⟩+≡
```
closest.map.list = function(xs, ys)
    lapply(xs, function(x) which.min(lapply(ys, function(y) eucl.dist(x,y))))
```

⟨*closest-points.py*⟩+≡
```
def closest_map_list(xs, ys):
    return [np.argmin([eucl_dist(x, y) for y in ys]) for x in xs]
```

Built-in functions are written in C; they hold a performance advantage over interpreted code, but are less flexible – they cannot be modified or extended by the user; and user code performing equivalent computations can not match their speed.

R's `colSums` and NumPy's `sum` are built-in functions that can produce a vector of results by summing each column (or row) of a two-dimensional array.

`eucl.dist.vec` is a vectorised version of `eucl.dist`, computing the distance between a single point `x` and a set of points `ys` in the columns of such an array. All the operations (`-`, `square`, `colsum`, and `sqrt`) are vectorised – the first two element-wise, recycling `x` over each column of `ys`. The resulting intermediate values are also arrays with the same dimension as `ys`, and are discarded immediately after use. In the R and NumPy versions, the cost of their allocation is paid in exchange for the benefit of calling the built-in functions.

In the language described here, the `colsum` function is not built in.

`closest.map.array` takes two sets of points stored in array format and performs a single iteration over the columns of `xs`, computing each index (via `which.min` again) of the closest point to `x` with a single call to `eucl.dist.vec` over all the `ys`.

⟨*closest-points.src*⟩+≡
```
const eucl.dist.vec = function(x, ys) : vector(double)
    sqrt(colsum(square(x - ys)))
const closest.map.array = function(array(double) xs,
                                   array(double) ys) : integer
    mapcols(xs, function(x) : int
                  which.min(eucl.dist.vec(x, ys)))
```

[1]Vaidya (1986) describes an optimal algorithm which builds and traverses a tree of boxes in $O(n \log n)$.

⟨*closest-points.R*⟩+≡

```
eucl.dist.vec = function(x, ys)
    sqrt(colSums((x - ys) ^ 2))
closest.map.array = function(xs, ys)
    apply(xs, 2, function(x) which.min(eucl.dist.vec(x, ys)))
```

⟨*closest-points.py*⟩+≡

```
def eucl_dist_vec(x, y):
    return np.sqrt(np.sum(np.power(x - y, 2), axis = 1))
def closest_map_array(xs, ys):
    return np.apply_along_axis(lambda x: np.argmin(eucl_dist_vec(x, ys)), 1, xs)
```

In the `closest.map` functions, only one entry is needed from the intermediate vector created for each `x`. In `closest.loop.list`, the explicit inner loop tracks the index `idx` of the current closest `y`, and so avoids allocating this vector.

⟨*closest-points.src*⟩+≡

```
const closest.loop.list = function(vector(numeric) xs,
                                   vector(numeric) ys) : integer
{
    var lx = length(xs), ly = length(ys), res = int(lx)
    for(i = 1; i <= lx; i = i + 1) {
        let idx = -1, d = Inf, x = xs[i]
        for(j = 1; j <= ly; j = j + 1) {
            let r = eucl.dist(x, ys[j])
            if(r < d) { idx = j; d = r; }
        }
        res[i] = idx
    }
    res
}
```

⟨*closest-points.R*⟩+≡

```
closest.loop.list = function(xs, ys) {
    res = integer(length(xs))
    i = 1
    for(x in xs) {
        d = Inf; idx = -1; j = 1
        for(y in ys) {
            r = eucl.dist(x, y)
            if(r < d) { idx = j; d = r; }
            j = j + 1
        }
        res[i] = idx
        i = i + 1
    }
    res
}
```

⟨*closest-points.py*⟩+≡
```
def closest_loop_list(xs, ys):
    res = np.zeros(len(xs), np.int)
    i = 0
    for x in xs:
        d = np.Inf; idx = -1; j = 0
        for y in ys:
            r = eucl_dist(x, y)
            if r < d: idx = j; d = r
            j += 1
        res[i] = idx
        i += 1
    return res
```

`closest.loop.array` takes its set of input points in the array representation instead, which is accomplished with only minor changes.

The columns of `xs` and `ys` are extracted into vectors by copying, except in the Python version – NumPy's `ndarray` indexing can produce an indirect 'view' of the same underlying buffer.

⟨*closest-points.src*⟩+≡
```
const closest.loop.array = function(array(double) xs,
                                    array(double) ys) : integer
{
    var lx = shape(xs)[2], ly = shape(ys)[2], res = int(lx)
    for(i = 1; i <= lx; i = i + 1) {
        let idx = -1, d = Inf, x = xs[,i]
        for(j = 1; j <= ly; j = j + 1) {
            let r = eucl.dist(x, ys[,j])
            if(r < d) { idx = j; d = r; }
        }
        res[i] = idx
    }
    res
}
```

⟨*closest-points.R*⟩+≡
```
closest.loop.array = function(xs, ys) {
    res = integer(ncol(xs))
    i = 1
    while(i <= ncol(xs)) {
        d = Inf; yn = -1; j = 1; x = xs[,i]
        while(j <= ncol(ys)) {
            r = eucl.dist(x, ys[,j])
            if(r < d) { idx = j; d = r; }
            j = j + 1
        }
        res[i] = idx
        i = i + 1
    }
    res
}
```

⟨*closest-points.py*⟩+≡

```
def closest_loop_array(xs, ys):
    res = np.zeros(xs.shape[0], np.int)
    for i in range(xs.shape[0]):
        d = np.Inf; idx = -1; x = xs[i,]
        for j in range(ys.shape[0]):
            r = eucl_dist(x, ys[j,])
            if r < d: idx = j; d = r
            j += 1
        res[i] = idx
        i += 1
    return res
```

These four alternative implementations are compared on test sets of size 1000, 2000 and 3000; each comprised of 100-dimensional points, generated randomly with `rnorm`.

⟨*closest-points.src*⟩+≡

```
const d = 100, r = 10
global xs, ys, xa, ya
const mklist = function(n) : vector(numeric)
    replicate(n, function():numeric rnorm(d))
const mkarr = function(n) : array(double) {
    let res = array(type(double), vec(d, n))
    res[] = rnorm(n*d)
    res
}
const test = function(fn)
    print(mean(replicate(r, function():double time(fn)[1])))
for(n = 1000; n <= 3000; n = n + 1000) {
    print(n)
    xs = mklist(n); ys = mklist(n); xa = mkarr(n); ya = mkarr(n)
    test(function() closest.map.list(xs, ys))
    test(function() closest.loop.list(xs, ys))
    test(function() closest.map.array(xa, ya))
    test(function() closest.loop.array(xa, ya))
}
```

⟨*closest-points.R*⟩+≡

```
d = 100; r = 10
prep = function(n, arr = FALSE)
    replicate(n, rnorm(d), simplify = arr)
test = function(fn)
    cat(mean(replicate(r, system.time(fn())[["elapsed"]])), "\n")
for(n in seq(1000, 3000, 1000)) {
    cat(n, "\n")
    xs = prep(n); ys = prep(n); xa = prep(n, T); ya = prep(n, T)
    test(function() closest.map.list(xs, ys))
    test(function() closest.loop.list(xs, ys))
    test(function() closest.map.array(xa, ya))
    test(function() closest.loop.array(xa, ya))
}
```

⟨*closest-points.py*⟩+≡
```
d = 100; r = 10
def mklist(n):
    return [np.random.normal(size = d) for x in range(0, n)]
def mkarr(n):
    return np.random.normal(size = (n, d))
def test(expr):
    times = timeit.repeat(expr, 'gc.enable()', number = 1,
                          repeat = r, globals=globals())
    print(math.fsum(times) / len(times))
for n in range(1000, 3000+1, 1000):
    print(n)
    xs = mklist(n); ys = mklist(n); xa = mkarr(n); ya = mkarr(n)
    test('closest_map_list(xs,ys)')
    test('closest_loop_list(xs,ys)')
    test('closest_map_array(xa,ya)')
    test('closest_loop_array(xa,ya)')
```

The average time taken by each function, over 10 executions, is recorded by `test`. Per language and input size, the results are shown in the following table (all times are in seconds, rounded to nearest 0.1s).

	`map.list`			`map.array`			`loop.list`			`loop.array`		
	1k	2k	3k	1k	2k	3k	1k	2k	3k	1k	2k	3k
Ours	**0.8**	**3.3**	**7.2**	0.9	4.5	12.5	**0.8**	**3.3**	**7.1**	**1.3**	**5.3**	**12.0**
R	2.7	11.7	27.3	**0.8**	**3.0**	**6.1**	2.2	8.7	20.3	4.5	18.2	41.1
Python	12.0	47.5	110.1	1.6	6.8	14.3	12.0	48.1	108.2	12.5	50.0	113.0

From this, some conclusions may be drawn. The use of built-in functions, as noted prior, make the `map.array` variant most performant in R and Python.

The latter is competitive with the language described here; the former faster by over a factor of 2. Its implementation of `colSums` has a specialised fast path for exactly this case (two-dimensional array of double-precision floating-point values, not treating `NA`s specially).

However, the costs the other languages pay for data not in the preferred format, or inner loops written in user code, are significant. The language described here also exhibits format preference – the `.array` variants allocate large temporary intermediate values, and the garbage collector is not tuned to the same extent as comparable implementations. Still, its performance differs by less than a factor of 2 between variants, versus over 6 for R, and over 7 for Python.

For all other variants, this language is faster, sometimes by a large margin – between 2 and 4 times compared to R; 10 and 12 times compared to Python. This reflects the efficacy of our chosen design – minor constraints on system flexibility in exchange for the applicability of powerful compiler techniques is a trade-off worth making.

1.3 Related Work

1.3.1 R

R can be considered a collection-oriented language, as its fundamental data type is the vector – scalars denote vectors of length one. The pervasive notion of "missing" values enables efficient, fine-grained treatment of data which may not be complete. It is dynamically typed, and implements an object model with multiple inheritance and multiple dispatch. Function arguments are lazily evaluated, allowing user control structures to be defined. Values are immutable, necessitating a copy-on-write strategy.

Originally evaluated by a tree-walking interpreter, R has been augmented to include a bytecode compiler and associated virtual machine (Tierney, 2001, 2016). While effective, the dynamic nature of the system (with such reflective primitives as `sys.frame` and `assign`) affords few static properties for an optimising compiler to exploit. Instead, the virtual machine has specialised fast paths for certain operations (such as scalar arithmetic), taken when applicable at run time. This improves performance on programs that use these features while preserving backward compatibility, at the cost of detecting such opportunities in the 'hot' inner loop of the interpreter.

In contrast, the language described here is explicitly designed for compilation, taking advantage of generic and type-directed optimisations. The virtual machine is a thin wrapper over the physical processor, being concerned primarily with storage management, residual type checking, and the calling convention.

Eager evaluation precludes using functions as control structures. A Lisp-style macro facility was envisaged as a replacement – evaluating user functions on the abstract syntax tree produced by the parser, before compiling the result. Such an extension is left to future work.

Call-by-reference semantics are simple, efficient, and familiar to users, but require the explicit copying of collections the user wishes to preserve before they are potentially mutated.

This is particularly salient at the top-level REPL, where functions that perform statistical analyses (for example) are not expected to modify the data sets upon which they are invoked.

Rigorous adherence to convention may suffice for a standard library, but future work may consider an explicit `immutable` annotation, to guard user functions against unexpectedly or erroneously mutating their input arguments.

In the absence of other side effects, immutability of values entails referential transparency (Strachey, 2000), which in turn renders applicable further powerful optimisations from the functional programming literature. While this line of research leads beyond the scope of this project, it is a potentially fruitful object of further work.

1.3.2 Python

Python (van Rossum, 1995) is a general-purpose interpreted programming language. It is dynamically typed and object-oriented, with single inheritance and single dispatch.

The language syntax provides for function parameter and return annotations, but they are not semantically meaningful in the reference implementation. Certain types of objects are inherently immutable, but there is no facility for the user to declare that certain of their functions or variables cannot be redefined or mutated. The language described here has type annotations and `const`ant declarations which are checked at compile-time (and enforced at run-time, where necessary).

The NumPy library (van der Walt et al., 2011) provides vector and array data structures for numerical and statistical programming. Specialised to machine scalar element types, these can be more efficient than the generic structures included in its standard library. The Pandas (McKinney, 2010) library provides a structure analogous to R's data frame.

CPython, the reference implementation of the language, compiles to bytecode without the benefit of any type-directed optimisations, unlike the language described in this document.

The Cython implementation (Behnel et al., 2011) is an ahead-of-time compiler which adds these, as well as type declarations, in order to compile to C. The PyPy implementation (Bolz et al., 2009) is a just-in-time trace compiler which generates machine instructions directly; its performance can be comparable to C.

1.3.3 Julia

Julia (Bezanson et al., 2012) is a recently developed programming language for numerical computing. It is dynamically typed, with optional static annotations and type recovery. Its object model features single inheritance and CLOS-style multimethods. Unusually, only classes without descendents may instantiate objects.

In sematics, it is closer to Matlab than to R. In design approach, it is similar to this work – dynamically typed, with optional annotations. Without run-time reflection mechanisms to preclude efficient compilation, the static properties of user programs can be exploited to produce optimised code.

Julia leverages the LLVM framework (Lattner and Adve, 2004) to perform some of these optimisations and, significantly, generate native machine instructions instead of interpreted bytecode. This yields performance comparable to C for general user code, rendering built-in functions unnecessary for many tasks.

The machinery necessary to generate and execute machine instructions has a significant implementation cost, especially if intended to be portable between platforms. It was passed over, in the system described here, in favour of exploring the interaction of language design and higher-level optimisations. Native code would be expected to perform up to an order of magnitude faster for certain input programs (Ertl and Gregg, 2003).

1.4 Future Work

- Multiprocessing VM, with synchronised access to global structures.
- Implement a generational or concurrent garbage collector, with a write barrier for references.
- Add more builtin functions – bindings for LAPACK, and more of Mathlib.
- Add heterogeneous container types: 'tuples' with ordered elements, 'structs' with named elements.
- Add an object model, with generic functions and multimethod dispatch.
- Replace the back-end of the compiler with an industrial-strength code generator, such as LLVM or GCC.
- Add macroexpansion to the compiler front-end.
- Investigate specialised optimisations for vector expressions, such as stream fusion.

1.5 Overview

The system may be decomposed into relatively independent subsystems, linked by the common data structures upon which they operate.

Parser (Chapter 4)

Parses *text* input according to the language grammar, creating an *abstract syntax tree.*

SSA Conversion (Chapters 6, 7, 8, 9)

Converts the *AST* to a graph-structured *intermediate representation*, which describes control and value flow in static single assignment form.

Optimisation (Chapters 10, 11, 12, 13, 14)

Analyses *IR* to recover types and constant values; then optimises the program by removing dead code, expanding function calls in-line, and eliminating redundant computations.

Code Generation (Chapters 15, 16, 17, 18)

Determines the lifetimes of *IR* values, allocates locations, generates *bytecode*.

Runtime (Chapters 19, 20, 21, 22, 23, 24, 25)

Provides services to the compiler, virtual machine, and user code: memory allocation, garbage collection, objects, types, builtin functions.

VM (Chapter 26)

A direct-threaded virtual machine interprets the *bytecode* instructions produced by the compiler, operating on *scalar* values and reference *objects*.

1.6 Licenses

Appendix A contains a C-program for MT19937-64 (2004/9/29 version), under the following license:

> Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura, All rights reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
>
> 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> 3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Miscellanea

⟨*lib*⟩≡

```
const retypeof = function(function f) ret_type(typeof(f))
const eltypeof = function(object v) elt_type(typeof(v))
time = function(function fn) {
    var a = rt_time(), v = fn(), b = rt_time()
    b - a
}
const replicate = function(int n, function fn, type etyp = retypeof(fn)) {
    let val = vector(etyp, n)
    for(i=1; i<=n; i=i+1)
        val[i] = fn()
    return val
}
const mean = function(vector x) sum(x) / length(x)
const square = function(x) x * x
const mapcols = function(array x, function fn, type etyp = retypeof(fn)) {
    var n = shape(x)[2], res = vector(etyp, n)
    for(i = 1; i <= n; i = i + 1)
        res[i] = fn(x[,i])
    res
}
const map = function(vector x, function fn, type etyp = retypeof(fn)) {
    var n = length(x), res = vector(etyp, n)
    for(i = 1; i <= n; i = i + 1)
        res[i] = fn(x[i])
    res
}
const which.min = function(vector xs) : int {
    var n = length(xs), m = 1, v = xs[1]
    for(i = 2; i <= n; i = i + 1) {
        var x = xs[i]
        if(x < v) {
            v = x
            m = i
        }
    }
    m
}
const colsum = function(x) mapcols(x, sum, eltypeof(x))
```

⟨*imports*⟩≡

```
import gc, timeit, math, numpy as np
```

Chapter 2

Executable

The system, composed of a compiler and run-time support module, is realised as a single monolithic executable (modulo the usual dynamically-loaded platform libraries such as the `libc`.)

⟨*main.c*⟩≡
⟨*includes*⟩
⟨*execute*⟩
⟨*evaluate*⟩
⟨*source*⟩
⟨*load*⟩
⟨*readline*⟩
⟨*init_default*⟩
⟨*init_name*⟩
⟨*load_init*⟩
⟨*repl*⟩
⟨*usage*⟩
⟨*main*⟩

Depending how the main executable is invoked, the system runs in one of two modes, *batch* or *interactive*.

The *runtime* subsystem (Chapter 19) is initialised before anything else; parts of its functionality are used extensively by the compiler itself.

⟨*main*⟩≡
```
int main(int argc, char *argv[])
{
    runtime_init();
    if(argc == 1)
        repl();
    else if(argc == 2 && argv[1][0] != '-')
        load(argv[1], true);
    else
        usage(argv[0]);
    runtime_fini();
    return 0;
}
```

If neither mode is applicable (perhaps the user has asked for `--help`,) a usage hint is printed.

⟨*usage*⟩≡
```
static void usage(char *name)
{
    fprintf(stderr, "usage:\t%s [file]\n", name);
    fprintf(stderr, "\tif file is not specified, INITFILE (default %s) is\n"
            "\tloaded if it exists, and the REPL is entered.\n", init_name());
    exit(-1);
}
```

2.1 Batch

In this mode, the system loads the given file and prints the result.

⟨*load*⟩≡
```
static void load(char *filename, bool print)
{
    evaluate(source(filename), print);
}
```

`source` reads and parses a file to an *abstract syntax tree*, `compile`s it to a callable `rclosure_t`, and returns the result. `c_begin` and `c_end` delimit the lifetime of the compiler; the latter also prints any errors or warnings encountered.

⟨*source*⟩≡
```
rclosure_t *source(char *filename)
{
    int r;
    rclosure_t *cl = NULL;
    ast_t ast;

    c_begin();
    r = p_source(filename, &ast);
    if(r == 0)
        cl = compile(&ast, filename);
    else if(r < 0)
        c_error("sourcing %s - %s", filename, strerror(errno));
    c_end();
    return cl;
}
```

2.2 Interactive

Interactive use is ideal for exploratory data analysis. Each iteration of the *read-evaluate-print loop* accepts an expression from the user. It's parsed, compiled, evaluated, and its result printed.

Before entry, the *init file*, if present, is loaded, and the `readline` library is initialised.

⟨*repl*⟩≡
```
static void repl()
{
    ast_t ast;
    int status;

    load_init();
    readline_init();
    c_begin();
    while((status = p_readline("> ", "+ ", &ast)) != -1)
    {
        rclosure_t *cl = NULL;
        if(status == 0)
            cl = compile(&ast, NULL);
        c_end();
        evaluate(cl, true);
        c_begin();
    }
    c_end();
    putchar('\n');
}
```

2.3 Evaluation

Successfully executing a closure results in an object – the value computed by the compiled expression. The runtime function `r_print` can then be used to output its printed representation, if requested. If an error occurs during execution, an error string is returned which needs `free`ing after printing.

⟨*evaluate*⟩≡
```
static void evaluate(rclosure_t *cl, bool print)
{
    void *res = NULL;

    if(!cl)
        return;
    if(execute(cl, &res))
    {
        putchar('\n');
        printf("%s", (char *)res);
        xfree(res);
    }
    else if(print)
    {
        putchar('\n');
        r_print(stdout, res);
    }
    putchar('\n');
}
```

The *virtual machine* is responsible for execution of compiled code. A `vm_ctx_t` context is initialised, passing the address and size of storage for the temporary stack. The VM (unlike the compiler) interoperates with the garbage collector, so the latter is enabled over its lifetime.

⟨*execute*⟩≡
```
uint8_t stack[65536];
static int execute(rclosure_t *cl, void **pres)
{
    vm_ctx_t ctx;

    vm_init_ctx(&ctx, stack, sizeof(stack));
    gc_set_enabled(true);
    if(vm_execute(&ctx, cl) == 0 && pres)
        *pres = ctx.ret_val;
    vm_fini_ctx(&ctx);
    gc_set_enabled(false);
    if(ctx.err_msg)
    {
        if(pres)
            *pres = ctx.err_msg;
        else
            xfree(ctx.err_msg);
        return -1;
    }
    return 0;
}
```

2.4 Init File

Users may want to enrich the global environment with their own code. They can create an init file which includes these additions. Its existence is checked so `source` won't issue an error should it be absent.

⟨*load_init*⟩≡
```
static void load_init()
{
    char *name = init_name();
    struct stat sbuf;

    if(stat(name, &sbuf) != 0)
        return;
    load(name, false);
}
```

If the `INITFILE` environment variable is set, it specifies the file to (attempt to) load in lieu of the default.

⟨*init_name*⟩≡
```
static const char *initvar = "INITFILE";
static char *init_name()
{
    char *name = getenv(initvar);
    return name ? name : init_default();
}
```

This is named "`.maininit`", in the user's home directory (in the unlikely case of running outside a `login` session, the current directory is used instead.)

⟨*init_default*⟩≡
```
static const char *initname = ".maininit";
static inline char *init_default()
{
    static char buf[PATH_MAX];
    char *base = getenv("HOME");

    if(!base)
        base = ".";
    snprintf(buf, PATH_MAX, "%s/%s", base, initname);
    return buf;
}
```

2.5 Readline

The interactive mode integrates the GNU `readline` library for line editing, command history and symbol completion.

At the prompt, pressing the "`TAB`" key (by default) will print a list of possible completions for the identifier at the point. This list is populated with the names of the variables in the global environment Section 19.5.

⟨*readline*⟩≡
⟨*compl_ctx_t*⟩
⟨*completions*⟩
⟨*complete_global*⟩
⟨*readline_init*⟩

⟨*readline_init*⟩≡
```
#define HISTORY_MAX 500
void readline_init()
{
    rl_initialize();
    rl_completion_entry_function = complete_global;
    rl_completer_quote_characters = "’‘";
    rl_completer_word_break_characters = " \t\n;(,)+-*/:<>={}[]";
    using_history();
    stifle_history(HISTORY_MAX);
}
```

⟨*compl_ctx_t*⟩≡
```
typedef struct
{
    const char *text;
    int len;
    ARRAY(char *) strs;
} compl_ctx_t;
```

⟨*complete_global*⟩≡
```
static char *complete_global(const char *text, int state)
{
    static int cur;
    static compl_ctx_t ctx;

    if(!state)
    {
        ctx = (compl_ctx_t) { text, strlen(text) };
        array_init(&ctx.strs, 1);
        hashmap_map(r_globals, completions, &ctx);
        cur = 0;
    }
    else
        cur++;
    if(cur >= alen(&ctx.strs))
    {
        array_fini(&ctx.strs);
        return NULL;
    }
    rl_completion_suppress_append = true;
    return strdup(aref(&ctx.strs, cur));
}
```

⟨*completions*⟩≡
```
static void completions(const void *key, void *value, void *ptr)
{
    compl_ctx_t *ctx = ptr;
    const rsymbol_t *name = key;
    char *str = name->string;

    if(!strncmp(str, ctx->text, ctx->len))
        array_push(&ctx->strs, str);
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "ast.h"
#include <errno.h>
#include <sys/stat.h>
#include <readline/readline.h>
#include <readline/history.h>
```

Part I

Compiler

Chapter 3

Compiler

The compiler subsystem supplies the `compile` entry point, which translates an abstract syntax tree into a callable object containing bytecode, ready for execution. A relatively straightforward direct-style compiler operating on an intermediate representation in static single assignment form, it has the peculiarity that it *recovers* types during optimisation, as opposed to *inferring* them or *checking* them beforehand.

⟨*compiler.c*⟩≡
```
⟨includes⟩
⟨globals⟩
⟨c_intern⟩
⟨c_messages⟩
⟨c_message_va⟩
⟨c_begin⟩
⟨c_end⟩
⟨pass prototypes⟩
⟨compile⟩
```

3.1 Entry Point

Compilation proceeds as a sequence of *passes*, each of which examines and perhaps transforms some or all of the program being compiled. The input is an abstract syntax tree from the parser (Chapter 4); a closure (Chapter 21) is returned that can be executed by the virtual machine (Chapter 26).

⟨*compile*⟩≡
```
rclosure_t *compile(ast_t *ast, char *name)
{
    cfunction_t *fn = NULL;
    rfunction_t *rfn = NULL;
    rclosure_t *cl = NULL;

    ⟨passes⟩
fail:
    if(fn)
        cfunc_free(fn);
    return cl;
}
```

The “front-end” of the compiler, `ir_convert`, converts the input `ast` into *intermediate representation*, which describes the control and data flow in the input program with a graph of pointers between compile-time objects (Chapter 5, Chapter 6). The resulting `fn` encapsulates the program for the remainder of the compilation process.

⟨*passes*⟩≡
```
fn = ir_convert(ast, name);
ast_fini(ast);
if(!fn)
    goto fail;
```

The program is cleaned, sorted and annotated by `ir_prepare` (Chapter 7).

⟨*passes*⟩+≡
```
if(failed(ir_prepare(fn)))
    goto fail;
```

Most local variables are erased the during *static single assignment conversion* performed by `ir_ssa_convert`, becoming edges in the value flow graph (Chapter 9).

⟨*passes*⟩+≡
```
ir_ssa_convert(fn);
```

The compiler’s “mid-end” is a mixture of necessary transformations and optional optimisations. The `ir_optimise` pass analyses the program using *sparse conditional constant propagation*, then uses the results to rewrite parts so that they are semantically equivalent but more efficient (Chapter 10, Chapter 11).

Lexically captured variables undergo *cell introduction* in `ir_cell_intro` (Chapter 12). Declared types are checked and any necessary conversions inserted by `ir_postpass`, which can also interpret certain complex argument lists ahead of time (Chapter 13).

A simple but effective *dominator value numbering* pass, implemented by `ir_dvn`, is capable of removing redundant computations (Chapter 14).

⟨*passes*⟩+≡
```
if(failed(ir_optimise(fn)))
    goto fail;
ir_cell_intro(fn);
if(failed(ir_postpass(fn)))
    goto fail;
ir_dvn(fn);
if(opt.dbg_dump_ir)
    ir_dump(fn);
```

Finally, the “back end” pass `gen_function` generates bytecode from the optimised intermediate representation, producing an `rfunction_t` object which is wrapped in a callable closure by `rcall_closure_create` (Chapter 15).

⟨*passes*⟩+≡
```
rfn = gen_function(fn);
cl = rcall_closure_create(rfn->cl_type, rfn);
```

3.2 Interface

If a pass encounters an error, `compile` must be informed so it can abandon the compilation. Some passes are run until convergence – they're iterated until no further changes are made. Some can avoid performing unnecessary operations if they know that a function had no effect.

To these ends, passes and their associated functions return `cresult` bitflags: to signal that they `CHANGED` something, or that they `FAILED`. `SUCCESS` is the neutral value.

⟨*cresult*⟩≡
```
typedef unsigned char cresult;
enum { SUCCESS = 0, FAILED = 1<<0, CHANGED = 1<<1 };
```

`cresult`s are bitwise-ORed together to accumulate flags across multiple callees, and tested by callers with the `failed` and `changed` predicates.

⟨*cresult predicates*⟩≡
```
static inline bool failed(cresult res) { return (res & FAILED) != 0; }
static inline bool changed(cresult res) { return (res & CHANGED) != 0; }
```

A small amount of state is kept over the lifetime of a compiler invocation – the list of diagnostic messages `c_msgs`, and the constant table `c_consttab`. Globals are bad form, but this subsystem is not intended to be re-entrant.

⟨*globals*⟩≡
```
static const char *lvlnames[] = { "Error", "Warning" };
static SLIST(cmessage_t) c_msgs;
static hashset_t *c_consttab;
```

They must be initialised before calling into the compiler.

⟨*c_begin*⟩≡
```
void c_begin()
{
    c_msgs = NULL;
    c_consttab = hashset_create(r_hash, r_equal);
}
```

After compilation, any messages that were queued are printed, then state is deallocated.

⟨*c_end*⟩≡
```
void c_end()
{
    hashset_free(c_consttab);
    c_messages(C_WARNING);
    assert(c_msgs == NULL);
}
```

Constant objects are built from literals in the input program, or generated by invocation of certain builtin functions with constant arguments.

They're internalised by the compiler in the same way that symbols and types are by the runtime.

⟨*c_intern*⟩≡
```
void *c_intern(void *ptr)
{
    void *obj = hashset_insert(c_consttab, ptr);
    if(obj != ptr)
        gc_release(ptr);
    return obj;
}
```

When an invocation of `compile` fails, an `ERROR` provides further information about the reason. A `WARNING` details an unexpected, probably undesired condition encountered during compilation.

⟨*cmsglevel*⟩≡
```
typedef enum { C_ERROR, C_WARNING } cmsglevel;
```

A message is represented by an instance of the `cmessage_t` structure. The `c_messages` list is linked through the `.next` field.

⟨*cmessage_t*⟩≡
```
typedef struct cmessage
{
    SLIST(struct cmessage) next;
    cmsglevel level;
    rsymbol_t *file;
    char *string;
} cmessage_t;
```

Other modules will call `c_message_va` to append a message to the global list...

⟨*c_message_va*⟩≡
```
void c_message_va(cmsglevel lvl, rsymbol_t *file, char *fmt, ...)
{
    va_list va;
    cmessage_t *msg = xcalloc(1, sizeof(*msg));

    va_start(va, fmt);
    msg->level = lvl;
    msg->file = file;
    if(vasprintf(&msg->string, fmt, va) == -1)
        fatal("vasprintf failed.");
    slist_push(c_msgs, msg, next);
    va_end(va);
}
```

...or use a shorthand macro.

⟨*message macros*⟩≡
```
#define c_warning(fmt, args...) c_message_va(C_WARNING, NULL, fmt, ##args)
#define c_error(fmt, args...) c_message_va(C_ERROR, NULL, fmt, ##args)
```

The `c_messages` function prints and clears the message list. A `level` argument of `C_ERROR` will muffle any warnings.

⟨*c_messages*⟩≡
```
void c_messages(int level)
{
    if(c_msgs)
    {
        cmessage_t *msg, *tmp;

        slist_nreverse(c_msgs, next);
        slist_foreach_safe(c_msgs, msg, tmp, next)
        {
            if(msg->level <= level)
                fprintf(stderr, "%s: %s\n", lvlnames[msg->level], msg->string);
            xfree(msg->string);
            xfree(msg);
        }
        c_msgs = NULL;
    }
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "ast.h"
```

Pass entry point prototypes are included inline, not from header files, as no other module needs to call them.

⟨*pass prototypes*⟩≡
```
cfunction_t *ir_convert(ast_t *ast, char *filename); // ir_convert.c
cresult ir_prepare(cfunction_t *fn); // ir_pre.c
cresult ir_postpass(cfunction_t *fn); // opt_post.c
void ir_init_closure(cfunction_t *fn); // ir_closure.c
void ir_ssa_convert(cfunction_t *fn); // ir_ssa.c
cresult ir_optimise(cfunction_t *fn); // opt_sccp.c
cresult ir_cell_intro(cfunction_t *fn); // opt_closure.c
void ir_dvn(cfunction_t *fn); // opt_dvn.c
rfunction_t *gen_function(cfunction_t *fn); // gen_code.c
```

⟨*compiler.h*⟩≡
 ⟨*cmsglevel*⟩
 ⟨*cmessage_t*⟩
 ⟨*cresult*⟩
 ⟨*cresult predicates*⟩
 ⟨*message macros*⟩
 ⟨*prototypes*⟩

⟨*prototypes*⟩≡
```
typedef struct cfunction cfunction_t;
cfunction_t *convert(ast_t *ast, char *name);
rclosure_t *source(char *name);
rclosure_t *compile(ast_t *ast, char *name);
void *c_intern(void *ptr);
void c_begin();
void c_end();
void c_messages(int level);
void c_message_va(cmsglevel lvl, rsymbol_t *file, char *fmt, ...);
```

Chapter 4

Parser

The parser subsystem is the first stage in the compiler, operating on textual input. We specify the language it is to recognise with a LALR(1) grammar; the GNU `bison` parser generator then gives us the `yyparse` function which forms the subsystem's core.

⟨*grammar.y*⟩≡

```
%code top {
    ⟨grammar includes⟩
}
%code requires {
    ⟨grammar.h⟩
}
%code {
    ⟨ast macros⟩
}
⟨declarations⟩
⟨terminals⟩
⟨precedence⟩
%%
⟨rules⟩
%%
⟨tok_empty⟩
⟨ast functions⟩
```

Assuming no errors are encountered, it creates from the input program an *abstract syntax tree*.

4.1 Abstract Syntax Tree

Each syntactic construct, when parsed, produces one or more instances of the `ast_t` structure. The `.type` tag determines which field in the union is populated.

⟨*ast_t*⟩≡

```
typedef struct ast ast_t;
typedef ARRAY(ast_t) ast_array_t;
typedef struct ast
{
    asttype type;
    union
    {
        ⟨ast union⟩
    };
} ast_t;
```

⟨*asttype*⟩≡
```
typedef enum
{
    AST_INVALID, AST_NODE, AST_TOKEN,
    AST_INT, AST_DOUBLE, AST_STRING,
    AST_QUOTED, AST_SYMBOL, AST_NAME
} asttype;
```

Values of the `asttype` enumeration are interpreted as follows.

`AST_NODE` Internal tree node, with array of `.children`. All other types are leaves.

⟨*ast union*⟩≡
```
ast_array_t *children;
```

`AST_TOKEN` A reserved identifier recognised by the parser. A value of `yytokentype` (from the generated `grammar.h`) is stored in the `.token` field.

⟨*ast union*⟩+≡
```
int token;
```

`AST_INT` Literal, stored in `.integer` field.
`AST_DOUBLE` Literal, stored in `.dfloat` field.
`AST_STRING` Literal, stored in `.string` field.

⟨*ast union*⟩+≡
```
int integer;
double dfloat;
char *string;
```

`AST_QUOTED` Literal (quoted) symbol.
`AST_SYMBOL` Symbolic identifier, probably a variable reference.
`AST_NAME` Name of a formal or actual argument.

These are all stored in the `.symbol` field. Symbols are garbage-collected strings, internalised in the runtime's symbol table (Section 19.2).

⟨*ast union*⟩+≡
```
rsymbol_t *symbol;
```

`AST_INVALID` Marks an omitted argument.

`ast_null` is a convenient source of this.

⟨*ast_null*⟩≡
```
ast_t ast_null = { .type = AST_INVALID };
```

Simple predicates are supplied for testing if a particular `ast_t` is the symbol "`...`", used for naming a "rest vector"...

⟨*ast_is_rest*⟩≡
```
static inline bool ast_is_rest(ast_t *ast)
    { return (ast->type == AST_SYMBOL && ast->symbol == r_sym_rest); }
```

...or an omitted argument marker.

⟨*ast_is_omitted*⟩≡
```
static inline bool ast_is_omitted(ast_t *ast)
    { return ast->type == AST_INVALID; }
```

Utility functions create ASTs of various shapes.

⟨*ast functions*⟩≡
⟨*ast_va*⟩
⟨*ast_set*⟩
⟨*ast_append*⟩
⟨*ast_prepend*⟩
⟨*ast_free_node*⟩
⟨*ast_fini*⟩

`ast_va` creates and returns a `NODE`, with children copied in order from its `NULL`-terminated vararg list of pointers.

⟨*ast_va*⟩≡
```
static ast_t ast_va(ast_t *head, ...)
{
    ast_array_t *arr;

    if(head)
    {
        va_list ap;
        ast_t *ptr;

        array_alloc(&arr, 2);
        array_push(arr, *head);
        va_start(ap, head);
        while((ptr = va_arg(ap, ast_t *)))
            array_push(arr, *ptr);
        va_end(ap);
    }
    else
        array_alloc(&arr, 0);
    return (ast_t) { .type = AST_NODE, .children = arr };
}
```

`ast_set` copies the given `ast` to the `i`th child of `expr`.

⟨*ast_set*⟩≡
```
static ast_t ast_set(ast_t *expr, ast_t *ast, int i)
{
    assert(expr->type == AST_NODE && expr->children);
    assert(alen(expr->children) > i);
    aset(expr->children, i, *ast);
    return *expr;
}
```

`ast_append` and `ast_prepend` add the given `ast` to the end and beginning of `expr`'s `.children`, respectively.

⟨*ast_append*⟩≡
```
static ast_t ast_append(ast_t *expr, ast_t *ast)
{
    assert(expr->type == AST_NODE && expr->children);
    array_push(expr->children, *ast);
    return *expr;
}
```

⟨*ast_prepend*⟩≡
```
static ast_t ast_prepend(ast_t *expr, ast_t *ast)
{
    assert(expr->type == AST_NODE && expr->children);
    array_insert(expr->children, 0, *ast);
    return *expr;
}
```

A `STRING` owns its `.string` value. A `NODE` owns its vector of `.children`...

⟨*ast_fini*⟩≡
```
void ast_fini(ast_t *ast)
{
    if(ast->type == AST_NODE)
        ast_free_node(ast->children);
    else if(ast->type == AST_STRING)
        xfree(ast->string);
}
```

...which must be recursively deallocated.

⟨*ast_free_node*⟩≡
```
static void ast_free_node(ast_array_t *arr)
{
    ast_t *ptr;

    array_foreach_ptr(arr, ptr)
        ast_fini(ptr);
    array_free(arr);
}
```

A handful of AST construction and mutation macros improve concision when writing rule actions.

⟨*ast macros*⟩≡
```
⟨prototypes⟩
#define AST1(a) ast_va(&a, NULL)
#define AST2(a, b) ast_va(&a, &b, NULL)
#define AST3(a, b, c) ast_va(&a, &b, &c, NULL)
#define AST4(a, b, c, d) ast_va(&a, &b, &c, &d, NULL)
#define ENDN(l, a, i) ast_set(&l, &a, i)
#define END1(l, a) ENDN(l, a, 0)
#define PUSH(l, a) ast_append(&l, &a)
#define PREP(l, a) ast_prepend(&l, &a)
```

4.2 Declarations

The `YYSTYPE` macro specifies the type of semantic value which grammar rules operate upon (and so the type of object ultimately returned by the parser.)

⟨*YYSTYPE*⟩≡
```
#define YYSTYPE ast_t
```

We use some extensions of `bison` over `yacc`. `api.pure` moves state from global variables to a function argument, making the generated parser re-entrant. `api.push-pull` is more interesting – it produces a `yypush_parse` function which can be invoked to parse one token at a time. This is ideal for interactive use (the original `yyparse` behaviour is provided via the wrapper `yypull_parse`.)

⟨*declarations*⟩≡
```
%define api.pure
%define api.push-pull both
```

Subsystem context is kept in a `pctx_t`; it's passed to the generated parser entry points, which in turn make it available to the lexer and rule actions.

⟨*declarations*⟩+≡
```
%parse-param {pctx_t *ctx}
%lex-param {pctx_t *ctx}
```

When recovering from a parse error, `bison` discards symbols and their semantic values from its internal stack. The latter may have allocated memory, so must free it.

When a parse succeeds, the initial symbol is discarded in the same way. Since its semantic value is desired result of the parse, it should not be discarded.

⟨*declarations*⟩+≡
```
%destructor { if(!ctx->done) ast_fini(&$$); } <>
%error-verbose // deprecated; should be %define parse.error verbose in 3.0+
```

4.2.1 Parser Context

⟨*pctx_t*⟩≡
```
typedef struct yypstate yypstate;
typedef struct pctx pctx_t;
typedef struct pctx
{
    ⟨parser context⟩
} pctx_t;
```

`bison`'s state is stored in the `.ps` field, merely to keep related objects together. When the parse succeeds, `.done` and the `.result` value are set.

⟨*parser context*⟩≡
```
yypstate *ps;
bool done;
ast_t result;
```

Interactive use requires distinguishing between partial and complete expressions. When a newline is input and the preceding expression is complete, it should be recognised and evaluated. When incomplete, a continuation prompt should be displayed so further input may be entered.

`.expr` is set by rules which parse complete expressions. `.nest` counts the depth of parenthesis, bracket and brace blocks. `.brk` is set when a newline is encountered in an incomplete expression.

⟨*parser context*⟩+≡
```
bool expr;
unsigned nest;
bool brk;
```

Interactive use also complicates solution of the "dangling `else`" problem. `.ifexpr` is set when an `if` expression has been parsed; `.ltok` and `.lval` then buffer a token's worth of lookahead in the case the next token lexed is not an `ELSE`.

⟨*parser context*⟩+≡
```
bool ifexpr;
int ltok;
ast_t lval;
```

`.buf` is a scratch buffer used by the lexer. `.get`, `.unget` and `.data` decouple the lexer from the details of stream access.

⟨*parser context*⟩+≡
```
char *buf;
int (*get)(pctx_t *);
void (*unget)(int, pctx_t *);
void *data;
```

4.2.2 Tokens

A *token* is a terminal symbol in the grammar. The name of a single character token is just that character. Tokens which match keywords or values are given names with `%token` and `%precedence`/`%left`/`%right` declarations.

⟨*terminals*⟩≡
```
%token LIT_INT LIT_DBL LIT_BOOL LIT_SYM LIT_STR ID TYPE FUNCTION
%token WHILE FOR BREAK CONTINUE LET VAR GLOBAL CONST INCLUDE LEXERR
```

Some terminals require precedence to be specified, to disambiguate shift/reduce conflicts between rules in which they appear. This list is ordered from lowest to highest. Function call (parenthesised argument list) and subscript (bracketed indices) have higher precedence than any operators; and assignment, lower.

⟨*precedence*⟩≡
```
%precedence RET
%precedence IF
%precedence ELSE
%right '='
%left OR AND
%nonassoc '>' GE '<' LE EQ NE
%left '&' '|'
%left '+' '-'
%left '*' '/' '%'
%left ':'
%precedence '!' UMINUS UPLUS
%left '^'
%precedence '[' '('
```

4.3 Lexical Analysis

The *lexer* `yylex` consumes characters from an input stream, producing one token per invocation. Each token can have an AST as its *semantic value*.

⟨*lexer.c*⟩≡
⟨*lexer includes*⟩
⟨*input functions*⟩
⟨*input macros*⟩
⟨*eat_ws*⟩
⟨*numbers*⟩
⟨*lex_id*⟩
⟨*lex_quoted*⟩
⟨*keywords*⟩
⟨*make_sym*⟩
⟨*make_str*⟩
⟨*make_delim*⟩
⟨*l_lex*⟩
⟨*yylex*⟩

`l_getc` and `l_ungetc` wrap the input functions in the parser context to track the current line number.

⟨*input functions*⟩≡
```
static inline int l_getc(pctx_t *ctx)
    { return ctx->get(ctx); }
static inline void l_ungetc(int c, pctx_t *ctx)
    { ctx->unget(c, ctx); }
```

Spaces and tabs are whitespace. A comment begins with a "#", extends to the end of the line (or the input), and is also considered whitespace. `eat_ws` consumes this until it encounters a non-whitespace character, returning the latter.

⟨*eat_ws*⟩≡
```
static int eat_ws(pctx_t *ctx)
{
    int c = l_getc(ctx);

    while(c == ' ' || c == '\t')
        c = l_getc(ctx);
    if(c == '#')
    {
        c = l_getc(ctx);
        while(c != '\n' && c != '\0' && c != EOF)
            c = l_getc(ctx);
    }
    return c;
}
```

Some terminals can have more than one value – numbers, identifiers, quoted strings. To lex one of these, one or more characters are copied from the input to a temporary buffer, as long as some condition holds. The `CPCHAR` and `CPCHARS` macros encapsulate this. They use the local variables `c`, `ptr`, `end`, and `ctx`.

⟨*input macros*⟩≡
```
#define CPCHARS(cond)                                     \
    for(; (cond) && c && ptr < end; c = l_getc(ctx))    \
        *((ptr)++) = c
#define CPCHAR(cond)                               \
    if((cond) && c && ptr < end)                   \
    {                                              \
        *((ptr)++) = c;                            \
        c = l_getc(ctx);                           \
    }
```

4.3.1 Lexer

The `l_lex` function is the core of the lexer. It returns one of the terminal tokens; if this has a useful semantic value, it will overwrite the `ast_null` initially stored at `out`.

Whitespace separates tokens, so `eat_ws` is called, and the returned character `c` used to distinguish what kind of token has been encountered.

⟨*l_lex*⟩≡
```
static int l_lex(YYSTYPE *out, pctx_t *ctx)
{
    int c = eat_ws(ctx);
    *out = ast_null;
    switch(c)
    {
    ⟨match char⟩
    default:
        break;
    }
    ⟨match op⟩
    ⟨match kwd⟩
    ⟨make token⟩
}
```

End-of-input markers are recognised by the grammar. `bison` puns single characters as tokens, so we can return them immediately.

⟨*match char*⟩≡
```
case '\0':
case EOF:
    return c;
```

A digit means a number must follow.

⟨*match char*⟩+≡
```
case '0' ... '9':
    return make_number(out, ctx, c);
```

One or more newlines count as a single token, regardless of whitespace.

⟨*match char*⟩+≡
```
case '\n':
    while((c = eat_ws(ctx)) == '\n');
    l_ungetc(c, ctx);
    return '\n';
```

Delimited tokens are strings which start and end with the same character.

Anything surrounded by acute accents "`" represents an identifier, as in R.

A string surrounded by apostrophes "'" denotes a literal symbol. Two symbols with the same name are, in fact, the same object, so compare `eq`.

A string surrounded by quotation marks """ is a literal string. These are not, at present, very useful – mostly for `print`ing, explaining why a program `stop`ped, and naming files to `include`.

⟨*match char*⟩+≡
```
case '`':
    return make_delim(out, ctx, '`', ID, make_sym);
case '\'':
    return make_delim(out, ctx, '\'', LIT_SYM, make_sym);
case '\"':
    return make_delim(out, ctx, '"', LIT_STR, make_str);
```

Some operators are two characters long. If `c` matches the start of one, check the next character. If that matches the next character of the same one, then that operator specifies the token we've lexed.

Otherwise, put the second char back; the first character denotes a different operator, so return one of those instead.

⟨*match op*⟩≡
```
int i = match_kwd_char(op2, lengthof(op2), c);
if(i != -1)
{
    int token = c;
    c = l_getc(ctx);
    if(c == op2[i].str[1])
        token = op2[i].token;
    else
        l_ungetc(c, ctx);
    return make_tok(out, token);
}
```

If `c` is an underscore, period or letter, it must be part of either a keyword or an identifier. Each keyword lexes to a different terminal. Identifiers lex to `ID`, with a `SYMBOL` as semantic value.

⟨*match kwd*⟩≡
```
if(c == '_' || c == '.' || isalpha(c))
{
    char *str = lex_id(ctx, c);
    int i = match_kwd_str(kwd, lengthof(kwd), str);
    if(i != -1) // keyword
        return make_tok(out, kwd[i].token);
    return make_sym(out, str, ID);
}
```

Otherwise, `c` is probably some single-character operator. Return it; the grammar can complain if it isn't recognised.

⟨*make token*⟩≡
```
return make_tok(out, c);
```

Terminals are not exclusively the grammar's concern. They'll be used later on by `ir_convert`, so carry `TOKEN`s `ast_t`s as semantic values.

⟨*make_tok*⟩≡
```
static inline int make_tok(YYSTYPE *out, int tok)
{
    *out = (ast_t) { .type = AST_TOKEN, .token = tok };
    return tok;
}
```

An identifier can contain digits, as well as underscores, periods and letters.

⟨*lex_id*⟩≡
```
static char *lex_id(pctx_t *ctx, int c)
{
    char *ptr = ctx->buf, *end = ctx->buf + L_BUFSIZE - 1;

    CPCHARS(isalnum(c) || c == '_' || c == '.');
    l_ungetc(c, ctx);
    assert(ptr < end);
    *ptr = '\0';
    return ctx->buf;
}
```

4.3.2 Numbers

Literal integers and double-precision floating-point reals are represented by the tokens `LIT_INT` and `LIT_DBL`, respectively.

⟨*numbers*⟩≡
⟨*make_integer*⟩
⟨*make_double*⟩
⟨*lex_number*⟩
⟨*make_number*⟩

Whether the digit `c` begins the former or the latter is determined by `lex_number`.

⟨*make_number*⟩≡
```
static int make_number(YYSTYPE *out, pctx_t *ctx, int c)
{
    int tok = lex_number(ctx, c);
    if(tok == LIT_DBL)
        return make_double(out, ctx);
    else if(tok == LIT_INT)
        return make_integer(out, ctx);
    return tok;
}
```

If a decimal point or exponent marker is encountered, it's a double, otherwise it's an integer. Digits (and these markers, if present) are copied into the context's scratch buffer `.buf`.

⟨*lex_number*⟩≡
```
static int lex_number(pctx_t *ctx, int c)
{
    char *ptr = ctx->buf, *end = ctx->buf + L_BUFSIZE - 1;
    int tok = LIT_INT;
    CPCHARS(isdigit(c));
    if(c == '.')
    {
        tok = LIT_DBL;
        CPCHAR(true);
        CPCHARS(isdigit(c));
    }
    if(c == 'e' || c == 'E')
    {
        tok = LIT_DBL;
        CPCHAR(true);
        CPCHAR(c == '+' || c == '-');
        CPCHARS(isdigit(c));
    }
    l_ungetc(c, ctx);
    assert(ptr < end);
    *ptr = '\0';
    return tok;
}
```

From there, an integer can be lexed with `strtol`...

⟨*make_integer*⟩≡
```
static int make_integer(YYSTYPE *out, pctx_t *ctx)
{
    long int val = 0;
    char *ptr;
    errno = 0;
    val = strtol(ctx->buf, &ptr, 10);
    if(ptr == ctx->buf)
    {
        yyerror(ctx, "can't parse integer literal");
        return LEXERR;
    }
    else if(errno == ERANGE || val > INT_MAX || val < INT_MIN)
    {
        yyerror(ctx, "integer literal out of range");
        return LEXERR;
```

```
    }

    *out = (ast_t) { .type = AST_INT, .integer = (int)val };
    return LIT_INT;
}
```

. . . or a double, with `strtod`.

⟨*make_double*⟩≡

```
static int make_double(YYSTYPE *out, pctx_t *ctx)
{
    double val = 0;
    char *ptr;
    errno = 0;
    val = strtod(ctx->buf, &ptr);
    if(ptr == ctx->buf)
    {
        yyerror(ctx, "can't parse floating-point literal");
        return LEXERR;
    }
    else if(errno == ERANGE)
    {
        yyerror(ctx, "floating-point literal out of range");
        return LEXERR;
    }

    *out = (ast_t) { .type = AST_DOUBLE, .dfloat = val };
    return LIT_DBL;
}
```

A `LEXERR` is returned instead if the literal value is out of range or otherwise invalid.

4.3.3 Quotation

Quoted identifiers, and literal symbols and strings, are all delimited tokens. Assuming an end `delim` is found, the `assign` function can create an `ast_t` from the copied characters and return the token.

⟨*make_delim*⟩≡

```
static inline int make_delim(YYSTYPE *out, pctx_t *ctx, char delim, int tok,
                             int (*assign)(YYSTYPE *, char *, int))
{
    if(!lex_quoted(ctx, delim))
        return LEXERR;
    return assign(out, ctx->buf, tok);
}
```

Within a delimited token, the backslash "\" introduces an escape sequence, but currently only newline is specially handled.

⟨*lex_quoted*⟩≡
```
static bool lex_quoted(pctx_t *ctx, char close)
{
    char *ptr = ctx->buf, *end = ctx->buf + L_BUFSIZE - 1;
    int c = l_getc(ctx);

    while(1)
    {
        CPCHARS(c != close && c != '\\');
        if(c == '\\')
        {
            c = l_getc(ctx);
            if(c=='n')
                c = '\n';
            CPCHAR(true);
            continue;
        }
        break;
    }

    if(c != close)
    {
        l_ungetc(c, ctx);
        return false;
    }
    assert(ptr < end);
    *ptr = '\0';
    return true;
}
```

A `SYMBOL` will probably be interpreted as a variable name, whereas when `QUOTED` it will denote itself, as a literal.

⟨*make_sym*⟩≡
```
static inline int make_sym(YYSTYPE *out, char *str, int tok)
{
    asttype type = (tok == ID) ? AST_SYMBOL : AST_QUOTED;
    *out = (ast_t) { .type = type, .symbol = r_intern(str) };
    return tok;
}
```

A `STRING` copies its value out of the scratch buffer to somewhere more permanent.

⟨*make_str*⟩≡
```
static inline int make_str(YYSTYPE *out, char *str, int tok)
{
    *out = (ast_t) { .type = AST_STRING, .string = strdup(str) };
    return tok;
}
```

4.3.4 Keywords and Operators

⟨*keywords*⟩≡
  ⟨*kwdspec_t*⟩
  ⟨*op2*⟩
  ⟨*kwd*⟩
  ⟨*match_kwd_char*⟩
  ⟨*match_kwd_str*⟩
  ⟨*make_tok*⟩
  ⟨*find_kwd*⟩
  ⟨*ast_str*⟩

A keyword specifier pairs a string with a token.

⟨*kwdspec_t*⟩≡
```
typedef struct
{
    const char *str;
    int token;
} kwdspec_t;
```

`op2` specifies the two-character-long operators.

⟨*op2*⟩≡
```
static const kwdspec_t op2[] = {
    { "&&", AND },
    { "||", OR },
    { "!=", NE },
    { "==", EQ },
    { ">=", GE },
    { "<=", LE },
};
```

`kwd` specifies the keywords.

⟨*kwd*⟩≡
```
static const kwdspec_t kwd[] = {
    { "type", TYPE },
    { "if", IF },
    { "else", ELSE },
    { "function", FUNCTION },
    { "return", RET },
    { "while", WHILE },
    { "for", FOR },
    { "break", BREAK },
    { "continue", CONTINUE },
    { "let", LET },
    { "var", VAR },
    { "global", GLOBAL },
    { "const", CONST },
    { "include", INCLUDE },
};
```

Helper functions match, against a table of keyword specifiers, a single character. . .

⟨*match_kwd_char*⟩≡
```
static int match_kwd_char(const kwdspec_t *tab, int len, int c)
{
    for(int i=0; i<len; i++)
        if(tab[i].str[0] == c)
            return i;
    return -1;
}
```

...or a string. The index of the matching specifier is returned; `-1` if none was found.

⟨*match_kwd_str*⟩≡

```
static int match_kwd_str(const kwdspec_t *tab, int len, char *str)
{
    for(int i=0; i<len; i++)
        if(!strcmp(tab[i].str, str))
            return i;
    return -1;
}
```

It is sometimes necessary to "unlex" an `ast_t` back to a string – currently, this only applies to `SYMBOL`s, `NAME`s and `TOKEN`s. This is easy enough for the first two; the last needs a search through the specifier tables for one with a matching `.token`.

⟨*ast_str*⟩≡

```
const char *ast_str(ast_t *ast)
{
    static char buf[2];

    assert(ast->type == AST_SYMBOL || ast->type == AST_NAME || ast->type == AST_TOKEN);
    if(ast->type == AST_SYMBOL || ast->type == AST_NAME)
        return r_symstr(ast->symbol);

    const char *str = find_kwd(op2, lengthof(op2), ast->token);
    if(str)
        return str;
    str = find_kwd(kwd, lengthof(kwd), ast->token);
    if(str)
        return str;
    buf[0] = ast->token;
    return buf;
}
```

The `find_kwd` helper performs this reverse lookup.

⟨*find_kwd*⟩≡

```
static const char *find_kwd(const kwdspec_t *tab, int len, int token)
{
    for(int i=0; i<len; i++)
        if(tab[i].token == token)
            return tab[i].str;
    return NULL;
}
```

4.3.5 Lexer Wrapper

Some 'features' of the language require collusion between the lexer and the grammar. `yylex`, the traditional `bison` entry point function, implements these workarounds.

⟨*yylex*⟩≡

```
int yylex(YYSTYPE *out, pctx_t *ctx)
{
    ⟨lexer wrapper⟩
}
```

If a lookahead token `.ltok` is present, it should be consumed instead of invoking `l_lex`.

⟨*lexer wrapper*⟩≡
```
int tok;
if(ctx->ltok != tok_empty)
{
    tok = ctx->ltok;
    *out = ctx->lval;
    ctx->ltok = tok_empty;
    ctx->lval = (ast_t){ 0 };
}
else
    tok = l_lex(out, ctx);
```

If, following an `if` expression and its terminating newline, an `else` is seen, it's passed through to the grammar. Otherwise a terminating "`;`" is synthesised, and the lexed token saved.

⟨*lexer wrapper*⟩+≡
```
if(ctx->ifexpr)
{
    if(tok != ELSE)
    {
        ctx->ltok = tok;
        ctx->lval = *out;
        tok = ';';
    }
    ctx->ifexpr = false;
}
```

Expressions can continue over several lines. If a newline can't be interpreted as terminating a "complete" expression (as signaled by the `.expr` flag in collusion with rule actions,) it should be skipped; the following token is returned by the tail call. Setting the `.brk` flag gives the interactive parser loop the opportunity to accept further input.

⟨*lexer wrapper*⟩+≡
```
if(tok == '\n')
{
    if(!ctx->expr)
    {
        ctx->brk = true;
        return yylex(out, ctx);
    }
}
ctx->expr = false;
return tok;
```

When a nonterminal that could end an expression is reduced, its rule sets `.expr` to signal that, should the following token be a newline, it's significant to the grammar and the lexer ought not to eat it.

Needless to say, this is frightfully ad-hoc and fragile – if `bison` needs a lookahead token to decide between shifting and reducing the rule, the action won't be able to write `.expr` until after the lexer's read it.

4.4 Grammar

The grammar is block-structured, curly-braced and expression-oriented, hewing close to R. It's fairly relaxed, in the sense that not all recognised inputs are legal programs (this is determined during conversion to IR, in Chapter 6.)

If the parser encounters an error, it notifies its caller. Otherwise, a list of expressions has been recognised. The `.done` flag is set and the `.result` updated – this `NODE` is the root of the tree; each child being a top-level expression.

⟨*rules*⟩≡
```
start:      exprlist { ctx->done = true; ctx->result = $1; }
    |       error { YYABORT; } ;
```

An expression list is a sequence of declaration expressions.

⟨*rules*⟩+≡
```
exprlist:   dexpr { $$ = AST1($1); }
    |       exprlist eos dexpr { $$ = PUSH($1, $3); }
    |       exprlist eos { $$ = $1; } ;
```

They are separated by newlines or semicolons. If the former is encountered while within a nested construct, more input is solicited from the interactive parser loop with `.brk`.

⟨*rules*⟩+≡
```
eos:        '\n' { if(ctx->nest > 0) ctx->brk = true; }
    |       ';' ;
```

The lexer consumes newlines only when it's certain it's safe to do so, passing them to the grammar in all other cases. If needed, they can then be ignored with the `nl` rule, which matches an optional newline.

⟨*rules*⟩+≡
```
nl:         { $$ = ast_null; } | '\n' { ctx->brk = true; } ;
```

Parentheses "`(`", brackets "`[`", and braces "`{`" surround nested constructs.

⟨*rules*⟩+≡
```
paren:      '(' { ctx->nest++; } ;
bracket:    '[' { ctx->nest++; } ;
brace:      '{' { ctx->nest++; } ;
```

A newline appearing before the closing delimiter of a parenthesised or bracketed construct can be ignored (but not braced, as it would conflict with a trailing `eos`.)

⟨*rules*⟩+≡
```
parclo:     nl ')' { ctx->nest--; } ;
brkclo:     nl ']' { ctx->nest--; } ;
braclo:     '}' { ctx->nest--; } ;
```

A declaration expression is a declaration list introduced with one of "`let`", "`var`", "`global`", or "`const`"; or an assignment expression.

⟨*rules*⟩+≡
```
dexpr:      aexpr
    |       LET decllist { $$ = AST2($1, $2); }
    |       VAR decllist { $$ = AST2($1, $2); }
    |       GLOBAL decllist { $$ = AST2($1, $2); }
    |       CONST decllist { $$ = AST2($1, $2); } ;
```

An assignment expression is just an expression; or an assignment "`=`", with an expression on the left-hand side and another assignment expression on the right-hand side (this is a separate rule because the same symbol is used to name actual arguments in a function call expression.)

⟨*rules*⟩+≡
```
aexpr:      expr %prec '=' { $$ = $1; }
    |       expr '=' aexpr { $$ = AST3($2, $1, $3); } ;
```

A parenthesised expression is just an **aexpr** inside parentheses.

⟨*rules*⟩+≡
```
  pexpr:      paren aexpr parclo { $$ = $2; } ;
```

An expression is an **if** expression, an **if** expression followed by an **else** expression, or a primary expression. This may end a complete expression, so the **.expr** flag is set upon reduction to signal the lexer.

⟨*rules*⟩+≡
```
  expr:       primary
              {
                  $$ = $1;
                  if(yychar == YYEMPTY || yychar == '\n')
                      ctx->expr = true;
              }
          |   ifexpr { $$ = $1; ctx->expr = true; }
          |   ifexpr elseexpr { $$ = PUSH($1, $2); ctx->expr = true; } ;
```

When an **if** expression is followed immediately by a newline within a nested construct, the lexer is informed via the **.ifexpr** flag, and the newline discarded.

The value of the **else** expression's **aexpr** is just appended to the preceding AST (since it'll always be an **ifexpr**.)

⟨*rules*⟩+≡
```
  ifexpr:     IF pexpr aexpr
              {
                  $$ = AST3($1, $2, $3);
                  if(yychar == '\n' && ctx->nest > 0)
                  {
                      ctx->ifexpr = true;
                      yyclearin;
                      ctx->brk = true;
                  }
              } ;
  elseexpr:   ELSE aexpr { $$ = $2; } ;
```

The primary expression rule comprises most other expressions in the language.

Identifiers, literals, parenthesised expressions for overriding precedence, unary and binary operations are all fairly conventional.

⟨*rules*⟩+≡
```
  primary:    ID
          |   literal
          |   pexpr
          |   block
          |   unary
          |   binary
```

A function call inserts the expression denoting the callee as the first child of the resulting **NODE**, in the manner of a LISP S-expression (McCarthy, 1960).

A subscript operation inserts the expression denoting the collection being indexed into as the second child; the first will be "**[**" (which may be pronounced "**aref**".)

⟨*rules*⟩+≡
```
          |   expr args { $$ = END1($2, $1); }
          |   expr subs { $$ = ENDN($2, $1, 1); }
```

A function expression has a list of formal arguments, an optional return type, and an **aexpr** body.

⟨*rules*⟩+≡
```
          |   FUNCTION formals rtopt nl aexpr { $$ = AST4($1, $3, $2, $5); }
```

A `while` loop takes a `pexpr` for its predicate, but a `for` loop requires a more complex list. Both also have `aexpr` bodies.

⟨*rules*⟩+≡
```
      |       WHILE pexpr aexpr { $$ = AST3($1, $2, $3); }
      |       FOR forlist aexpr { PUSH($2, $3); $$ = END1($2, $1); }
```

`break`, `continue` and `return` transfer control flow, and should probably be considered statements by the grammer (IR conversion does so in its stead.)

⟨*rules*⟩+≡
```
      |       RET expr { $$ = AST2($1, $2); }
      |       BREAK
      |       CONTINUE
```

The file named by the argument of `include` will be read and parsed during IR conversion, its AST taking the place of the expression in the including program.

⟨*rules*⟩+≡
```
      |       INCLUDE LIT_STR { $$ = AST2($1, $2); } ;
```

A lexed literal token is a literal, as is a type expression – the terminal "`type`" followed by a type specifier in parentheses.

⟨*rules*⟩+≡
```
  literal:    LIT_INT | LIT_DBL | LIT_BOOL | LIT_STR | LIT_SYM
      |       TYPE paren type parclo { $$ = AST2($1, $3); } ;
```

A block is a brace-delimited list of expressions. Unlike a top-level `exprlist`, the `NODE` created begins with a "`{`" terminal.

⟨*rules*⟩+≡
```
  block:      brace exprlist braclo { $$ = PREP($2, $1); }
      |       brace braclo { $$ = AST1($1); } ;
```

The application of a unary operator takes an expression as its operands (because `bison` runs before `cpp`, macros involving the `$n` semantic variables can't be defined nicely outside.)

The operator token is placed first in the `NODE`, like a function call.

⟨*rules*⟩+≡
```
  unary:      '!' expr
              {
                  #define UNOP $$ = AST2($1, $2)
                  UNOP;
              }
      |       '-' expr %prec UMINUS { UNOP; }
      |       '+' expr %prec UPLUS  { UNOP; }
```

Binary operators operate in a similar manner.

⟨*rules*⟩+≡
```
binary:     expr ':' expr
            {
                #define BINOP $$ = AST3($2, $1, $3)
                BINOP;
            }
    |       expr '+' expr { BINOP; }
    |       expr '-' expr { BINOP; }
    |       expr '*' expr { BINOP; }
    |       expr '/' expr { BINOP; }
    |       expr '^' expr { BINOP; }
    |       expr '%' expr { BINOP; }
    |       expr '&' expr { BINOP; }
    |       expr '|' expr { BINOP; }
    |       expr '<' expr { BINOP; }
    |       expr LE expr  { BINOP; }
    |       expr EQ expr  { BINOP; }
    |       expr NE expr  { BINOP; }
    |       expr GE expr  { BINOP; }
    |       expr '>' expr { BINOP; }
    |       expr AND expr { BINOP; }
    |       expr OR expr  { BINOP; } ;
```

The return type of a function is, if specified, separated from the argument list by a "`:`" terminal. Formal arguments themselves are optional, but the enclosing parentheses are not.

⟨*rules*⟩+≡
```
rtopt:      { $$ = ast_null; } | ':' type { $$ = $2; } ;
formals:    paren decllist parclo { $$ = $2; }
    |       paren parclo { $$ = ast_va(NULL); } ;
```

A `for` loop takes, in order, an optional list of declarations, a conditional, and an optional iteratior, enclosed in parentheses and separated by semicolons.

⟨*rules*⟩+≡
```
forlist: paren dlopt ';' aexpr ';' aexopt parclo
    { $$ = AST4(ast_null, $2, $4, $6); } ;
dlopt:      { $$ = ast_null; }
    |       decllist ;
aexopt:     { $$ = ast_null; }
    |       aexpr ;
```

Lists of actual arguments and subscripts differ only in the brackets they're enclosed in.

The empty `args` and `subs` can't consume an optional newline without introducing ambiguity, so must mention their closing delimiter (and reduce the `.nest` count) explicitly.

⟨*rules*⟩+≡
```
args:       paren arglist parclo { $$ = $2; }
    |       paren ')' { $$ = AST1(ast_null); ctx->nest--; } ;
subs:       bracket arglist brkclo { $$ = PREP($2, $1); }
    |       bracket ']' { $$ = AST2($1, ast_null); ctx->nest--; } ;
```

Each argument in the comma-separated list can be omitted (its element in the call's `NODE` will be `INVALID`) or preceded by a `name` "`=`" (the name likewise precedes the argument expression's element in the call.)

⟨*rules*⟩+≡
```
arglist:    arg { $$ = AST2(ast_null, $1); }
       |    name '=' expr { $$ = AST3(ast_null, $1, $3); }
       |    arglist ',' arg { $$ = PUSH($1, $3); }
       |    arglist ',' name '=' expr { PUSH($1, $3); $$ = PUSH($1, $5); } ;
arg:        { $$ = ast_null; } | expr ;
```

A `name` is lexed as an identifier, but parsed with a type of `NAME` instead of `SYMBOL`, so IR conversion can distinguish otherwise ambiguous AST sequences.

⟨*rules*⟩+≡
```
name:       ID { $$ = $1; $$.type = AST_NAME; ctx->expr = true; } ;
```

Declaration lists are comma-separated and non-empty. A declaration comprises a name, which may be preceded by an optional type specifier and followed by an optional initial expression.

⟨*rules*⟩+≡
```
decllist:   decl { $$ = AST1($1); }
       |    decllist ',' decl { $$ = PUSH($1, $3); } ;
decl:       name
       |    name initexpr { $$ = AST2($1, $2); }
       |    type name { $$ = AST2($1, $2); }
       |    type name initexpr { $$ = AST3($1, $2, $3); } ;
initexpr:   '=' expr { $$ = $2; } ;
```

Types are either named or constructed. `type` and `function` are type names but also terminal symbols, so don't lex as `name`s and must be matched separetely. Constructor invocation follows the name with a parenthesis-enclosed comma-separated list of type arguments.

The `function` type also allows specification of return type in the same manner as a function expression (it ends up the second child in the `NODE`, and is `INVALID` if omitted.)

⟨*rules*⟩+≡
```
type:       name
       |    name typeargs { $$ = END1($2, $1); }
       |    TYPE
       |    FUNCTION
       |    FUNCTION typeargs { $$ = PREP($2, $1); }
       |    FUNCTION typeargs ':' type { END1($2, $4); $$ = PREP($2, $1); } ;
typeargs:   paren typelist parclo { $$ = $2; }
       |    paren parclo { $$ = AST1(ast_null); } ;
typelist:   type { $$ = AST2(ast_null, $1); }
       |    typelist ',' type { $$ = PUSH($1, $3); } ;
```

4.5 Parser Interface

Connecting input sources to the lexer and generated `yyparse` functions is the purpose of the `parser.c` module.

⟨*parser.c*⟩≡
  ⟨*parser includes*⟩
  ⟨*ast_null*⟩
  ⟨*file_getc*⟩
  ⟨*file_ungetc*⟩
  ⟨*linedata_t*⟩
  ⟨*line_getc*⟩
  ⟨*line_ungetc*⟩
  ⟨*pctx_init*⟩
  ⟨*pctx_fini*⟩
  ⟨*read_line*⟩
  ⟨*parse_line*⟩
  ⟨*p_readline*⟩
  ⟨*p_file*⟩
  ⟨*p_source*⟩
  ⟨*yyerror*⟩

`stdio` handles buffering when reading from files. The `FILE` pointer is stashed in the parser context's `.data` field.

⟨*file_getc*⟩≡
```
static int file_getc(pctx_t *ctx)
    { return getc(ctx->data); }
```

⟨*file_ungetc*⟩≡
```
static void file_ungetc(int c, pctx_t *ctx)
    { ungetc(c, ctx->data); }
```

For a line read interactively, we keep the string `.str`, a cursor `.ptr`, and the limit `.end`.

⟨*linedata_t*⟩≡
```
typedef struct
{
    char *str, *ptr, *end;
} linedata_t;
```

The string returned by `readline` ends in a `NUL` byte, although the user typed a newline. We undo this. Reading past the end of the line returns `EOF`.

⟨*line_getc*⟩≡
```
static int line_getc(pctx_t *ctx)
{
    linedata_t *data = ctx->data;
    int c = EOF;

    if(data->ptr < data->end)
        c = *data->ptr++;
    if(c == '\0')
        c = '\n';
    return c;
}
```

Once this occurs, the stream has ended, and further reads will always return `EOF` (no error checking here; the lexer oughtn't `.ungetc` with a value not returned by the previous `.getc`.)

⟨*line_ungetc*⟩≡
```
static void line_ungetc(int c, pctx_t *ctx)
{
    linedata_t *data = ctx->data;

    if(c != EOF)
        data->ptr--;
}
```

Parser context is initialised by setting the input fields, allocating a new `bison` state and temporary lexer buffer.

⟨*pctx_init*⟩≡
```
static void pctx_init(pctx_t *ctx, int (*get)(pctx_t *),
                      void (*unget)(int, pctx_t *), void *data)
{
    *ctx = (pctx_t) {
        .ps = yypstate_new(),
        .result = ast_null,
        .buf = xmalloc(L_BUFSIZE),
        .ltok = tok_empty,
        .get = get,
        .unget = unget,
        .data = data
    };
}
```

(which must be deallocated when no longer needed.)

⟨*pctx_fini*⟩≡
```
static void pctx_fini(pctx_t *ctx)
{
    yypstate_delete(ctx->ps);
    xfree(ctx->buf);
}
```

The generated parser calls `yyerror` when a syntax error is encountered; its message is forwarded to the compiler for later display.

⟨*yyerror*⟩≡
```
void yyerror(pctx_t *ctx, const char *err)
{
    c_error("%s", err);
}
```

Parsing a file initialises the context with `.nest` nonzero – the newline following an `if` expression's consequent mustn't be interpreted as ending the `if` without first looking ahead for an `else`.

`yypull_parse` calls `yylex` repeatedly, constructs the AST and, if successful, returns zero. After extracting the desired `.result`, so do we.

If unsuccessful, this must be freed – it's possible for some `exprlist` to have reduced to `start` (assigning `.result`), then an unexpected token cause a syntax error.

⟨*p_file*⟩≡
```
static int p_file(FILE *input, ast_t *result)
{
    int status;
    pctx_t ctx;

    pctx_init(&ctx, file_getc, file_ungetc, input);
    ctx.nest = 1;
    status = yypull_parse(ctx.ps, &ctx);
    if(status == 0)
        *result = ctx.result;
    else
        ast_fini(&ctx.result);
    pctx_fini(&ctx);
    return status;
}
```

A convenience wrapper handles the `FILE`, allowing users to specify a filename instead.

⟨*p_source*⟩≡
```
int p_source(char *name, ast_t *result)
{
    int r;
    FILE *fp = fopen(name, "r");

    if(fp)
    {
        r = p_file(fp, result);
        fclose(fp);
    }
    else
        r = -errno;
    return r;
}
```

Parsing interactive input is done a line at a time. The parser context is initialised, and an initial prompt `p1` is displayed. A line is read from the terminal; if successful (i.e. the user didn't input `EOF`) the line is parsed and the prompt changed to the continuation `p2`. This goes on until the user quits, or the parser encounters a syntax error, or the start nonterminal is recognised.

⟨*p_readline*⟩≡
```
int p_readline(char *p1, char *p2, ast_t *result)
{
    linedata_t data;
    pctx_t ctx;
    int status;
    char *prompt = p1;

    pctx_init(&ctx, line_getc, line_ungetc, &data);
    do
```

```
    {
        status = read_line(&ctx, prompt);
        if(status != 0)
            break;
        status = parse_line(&ctx, prompt == p1);
        prompt = p2;
    } while(status == YYPUSH_MORE);
    if(status == 0)
        *result = ctx.result;
    else
        ast_fini(&ctx.result);
    pctx_fini(&ctx);
    return status;
}
```

`readline` prints the `prompt` and handles terminal input, returning `NULL` on `EOF`. If the `line` isn't blank, we add it to the input history. The `linedata_t` is initialised with the cursor at the beginning, and the limit set to encompass the length of the string plus the terminating `NUL`.

⟨*read_line*⟩≡

```
static int read_line(pctx_t *ctx, char *prompt)
{
    linedata_t *data = ctx->data;
    char *line = readline(prompt);

    rl_free_undo_list();
    if(!line)
        return -1;
    if(*line)
        add_history(line);
    *data = (linedata_t) {
        .str = line,
        .ptr = line,
        .end = line + strlen(line) + 1
    };
    return 0;
}
```

Tokens are `yylex`ed from the line buffer and fed to `yypush_parse` one at a time, as it continues to return `YYPUSH_MORE`. `.brk` has the opportunity to interrupt input after either step. If the user's initial line is blank, we return with `status` 1 to quit the outer loop.

Whichever way the inner loop ends, the input string can now be discarded.

⟨*parse_line*⟩≡

```
static int parse_line(pctx_t *ctx, bool isfirst)
{
    linedata_t *data = ctx->data;
    int status = isfirst ? 1 : YYPUSH_MORE;

    ctx->brk = false;
    do
    {
        ast_t val;
        int tok = yylex(&val, ctx);
        if(!ctx->brk)
            status = yypush_parse(ctx->ps, tok, &val, ctx);
    } while(!ctx->brk && status == YYPUSH_MORE);
```

```
    xfree(data->str);
    return status;
}
```

Miscellanea

⟨*grammar includes*⟩≡
```
#include "global.h"
#include "ast.h"
#include "grammar.h"
```

⟨*prototypes*⟩≡
```
static ast_t ast_va(ast_t *head, ...);
static ast_t ast_set(ast_t *expr, ast_t *ast, int i);
static ast_t ast_append(ast_t *expr, ast_t *ast);
static ast_t ast_prepend(ast_t *expr, ast_t *ast);
```

⟨*tok_empty*⟩≡
```
const int tok_empty = YYEMPTY;
```

⟨*lexer includes*⟩≡
```
#include "global.h"
#include "ast.h"
#include "grammar.h"
#include <ctype.h>
#include <limits.h>
#include <errno.h>
```

⟨*parser includes*⟩≡
```
#include "global.h"
#include "ast.h"
#include "grammar.h"
#include <signal.h>
#include <errno.h>
#include <readline/readline.h>
#include <readline/history.h>
```

⟨*ast.h*⟩≡
⟨*asttype*⟩
⟨*ast_t*⟩
⟨*ast_is_rest*⟩
⟨*ast_is_omitted*⟩
⟨*ast prototypes*⟩

⟨*ast prototypes*⟩≡
```
void ast_fini(ast_t *ast);
const char *ast_str(ast_t *ast);
int p_source(char *name, ast_t *ast);
int p_readline(char *p1, char *p2, ast_t *result);
```

⟨*grammar.h*⟩≡
⟨*YYSTYPE*⟩
⟨*pctx_t*⟩
⟨*grammar externs*⟩
⟨*L_BUFSIZE*⟩
⟨*grammar prototypes*⟩

⟨*grammar externs*⟩≡
```
extern ast_t ast_null;
extern const int tok_empty;
```

⟨*L_BUFSIZE*⟩≡

```
#define L_BUFSIZE 256
```

⟨*grammar prototypes*⟩≡

```
int yylex(YYSTYPE *out, pctx_t *ctx);
void yyerror(pctx_t *ctx, const char *err);
```

Chapter 5

Intermediate Representation

The abstract syntax tree has the shape of the program as written by the user. To enable further analysis, it is converted to an *intermediate representation* having the shape of the program as it will execute (although it's translated into bytecode – a more concise and efficient format – before actually being run.)

⟨*ir.h*⟩≡
  ⟨*preliminaries*⟩
  ⟨*cfunction_t*⟩
  ⟨*cvartype*⟩
  ⟨*cvar_t*⟩
  ⟨*cblock_t*⟩
  ⟨*cnodetype*⟩
  ⟨*cnode_t*⟩
  ⟨*prototypes*⟩
  ⟨*inlines*⟩

5.1 Structures

A program in intermediate representation consists of a tree of *functions*. Each function contains a directed graph of basic *blocks*; each block, a list of *nodes*.

Control flow within a function follows the edges between blocks; the flow of computed values, edges between nodes.

Variables stand in for values supplied by actual arguments, or the global or lexical environment.

These structures have compilation lifetime – they do not persist beyond a single invocation of the compiler, and it explicitly manages their storage. The runtime does not interact with them.

5.1.1 Functions

One `cfunction_t` is created for each "`function`" expression in the program source. Neither control nor value flow cross function boundaries.

⟨*cfunction_t*⟩≡
```
typedef struct cfunction
{
    ⟨function⟩
} cfunction_t;
```

A backpointer to the (`cnode_t` created from the) expression is kept in the `.node` field. `.parent` points from enclosed to enclosing function in the tree induced by lexical inclusion.

An expression outside any function is at *top level*; a distinguished `cfunction_t` is created to contain these. It is at the root of the function tree, with `.parent` and `.node` both `NULL`.

⟨*function*⟩≡
```
cfunction_t *parent;
cnode_t *node;
```

When a function expression is evaluated, a closure (callable object) is created. The number, order, and declared types of the function's formal arguments, as well as its return type, determine the type of these closures, held in the `.cl_type` field (Chapter 21).

⟨*function*⟩+≡
```
rtype_t *cl_type;
```

Enclosed functions are kept on a list headed by `.cfunc_head` in the parent and threaded through the `.cfunc_list` of each child. Variables which are bound by the function – locals and arguments; and globals at the top level – are on the list headed by `.cfunc_head`. The basic blocks comprising the function's body are on the list headed by `.cblock_head`. `.entry` points to the block which begins the function body.

⟨*function*⟩+≡
```
list_t cfunc_list, cfunc_head;
list_t cvar_head, cblock_head;
cblock_t *entry;
```

Each function has an `.id` number different from its siblings; and has `.nfuncs` child functions, `.nblocks` blocks, and `.nnodes` nodes in its body. When functions are inlined into this one, their nodes are counted in `.ninlined`.

⟨*function*⟩+≡
```
unsigned id, nfuncs, nblocks, nnodes;
unsigned ninlined;
```

Closure analysis discovers which variables free in the function body are bound by lexical ancestors. Pointers to them are stored in the `.closure` array.

⟨*function*⟩+≡
```
cvar_array_t closure;
```

5.1.2 Variables

A variable denotes a named quantity. They can be local to a function, defined at top level in the input program, or read from the global environment – a map of names to values, managed by the runtime (Section 19.5).

⟨*cvartype*⟩≡
```
typedef enum
{
    LEXICAL, GLOBAL_INT, GLOBAL_EXT,
} cvartype;
```

When a variable is first introduced, a `cvar_t` is created.

⟨*cvar_t*⟩≡
```
typedef struct cvar
{
    ⟨var⟩
} cvar_t;
```

If local, `.cvar_list` links the variable into its binding function's list; otherwise, the top-level function takes possession.

⟨*var*⟩≡
```
list_t cvar_list;
```

`.name` holds the symbol which names the variable. `.decl` holds the type present at its declaration, or `NULL` if none was supplied.

⟨*var*⟩+≡
```
rsymbol_t *name;
rtype_t *decl;
```

A local variable may be inferred *constant* if it is never assigned to after being initialised (and a global variable declared constant with a "`const`" expression.) The `.is_const` will be set in this case.

⟨*var*⟩+≡
```
bool is_const;
```

Variables are assigned an `.id` number, unique for all those bound by the same function. A pass may compare `.mark` with a global clock to determine if it's encountered a variable before.

⟨*var*⟩+≡
```
unsigned id, mark;
```

The compiler may have more or less information about a variable, depending on how it comes to be known. The former is stored in a union, tagged with a `.type` representing the latter.

⟨*var*⟩+≡
```
cvartype type;
union
{
    ⟨var union⟩
};
```

`LEXICAL` variables are local to a function.

They're either formal arguments, or introduced in its body with "`let`" or "`var`" forms. Their scope is completely visible to the compiler.

`.binder` holds a backpointer to the function, and `.cvar_list` is linked into its variable list.

A formal argument has the `.is_arg` flag set. It can also be specified with a default expression; `.is_optional` is set if this is the case. When the corresponding actual argument is omitted from a call, the variable takes its value from the default expression instead (Subsection 6.6.4).

If the variable occurs free in the body of a descendent function, it will be need to be captured by at least one closure, and `.is_closed` will be set.

⟨*var union*⟩≡
```
struct
{
    cfunction_t *binder;
    bool is_arg;
    bool is_optional;
    bool is_closed;
} local;
```

GLOBAL_INT variables are declared by the input program.

They are introduced with "global" or "const" forms. An internal global's initialising expression .set is visible to the compiler.

⟨*var union*⟩+≡
```
struct
{
    cnode_t *set;
} intl;
```

GLOBAL_EXT variables are part of the runtime global environment.

The compiler may not necessarily rely on any of their properties (or even their existence, in some cases.)

Such a variable may have been declared by code executed beforehand; if so, .global points to its binding. If it wasn't found at compile time, this field is NULL; it may be implicitly created by an assignment later on.

⟨*var union*⟩+≡
```
struct
{
    rglobal_t *global;
} extl;
```

5.1.3 Blocks

The unit of control flow within a function is the cblock_t: an extended basic block (Aho et al., 2006, Section 8.4).

⟨*cblock_t*⟩≡
```
typedef struct cblock
{
    ⟨block⟩
} cblock_t;
```

.cblock_list links into the list of its containing function. In later passes, this is sorted in reverse-depth-first order.

⟨*block*⟩≡
```
list_t cblock_list;
```

Each block contains a list of nodes, in control-flow order from .cnode_head.

⟨*block*⟩+≡
```
list_t cnode_head;
```

Control flow within a function begins with the first node in its .entry block. Every other block has one or more predecessors, pointers to which are stored in the .pred array.

⟨*block*⟩+≡
```
cblock_array_t pred;
```

The final node in a block may return from the function. Execution may continue to another block, or branch to one of two alternatives depending on a boolean condition. In the latter cases, pointers to the successor blocks are stored in the .succ array.

⟨*block*⟩+≡
```
cblock_array_t succ;
```

Blocks are also assigned `.id` numbers (as per the list, in reverse-depth-first order.) After numbering, `.start` and `.end` will contain the `.id`s of the first and last nodes in the block. The `.mark` field serves the same purpose as `cvar_t`'s.

⟨*block*⟩+≡
```
unsigned id, start, end, mark;
```

5.1.4 Nodes

Nodes represent computations, the various kinds of which are enumerated by `cnodetype`.

⟨*cnodetype*⟩≡
```
typedef enum
{
    CN_LAMBDA, CN_CALL, CN_CALL_FAST, CN_BUILTIN,
    CN_SET, CN_BIND, CN_REF, CN_IF, CN_RETURN,
    CN_CONST, CN_COPY, CN_PHI
} cnodetype;
```

Most expressions in a program create one or more `cnode_t`s. A node may produce a value that following nodes can use as input.

⟨*cnode_t*⟩≡
```
typedef struct cnode
{
    ⟨node⟩
} cnode_t;
```

Each node is linked into its containing `.block`'s list via the `.cnode_list` field.

⟨*node*⟩≡
```
list_t cnode_list;
cblock_t *block;
```

The type of value a node produces might be known ahead of time. Some nodes will be assigned a type by user declaration; as analysis proceeds, others can have theirs recovered. If `.decl` is non-`NULL`, the compiler has determined that `r_subtypep(`*value*`, .decl)` is a tautology.

⟨*node*⟩+≡
```
rtype_t *decl;
```

The `.users` array contains pointers to the nodes which depend on the value of this one. It's populated early in compilation, and kept up-to-date thereafter.

⟨*node*⟩+≡
```
cnode_array_t users;
```

Nodes are numbered along with blocks. `.id` is unique within the function, increases in reverse-depth-first order between blocks and control-flow (i.e. list) order within them. The `.mark` field is the same as for `cblock_t` and `cvar_t`.

⟨*node*⟩+≡
```
unsigned id, mark;
rsymbol_t *file;
```

A node's `.type` determines its behaviour, and tags a branch of the union.

⟨*node*⟩+≡
```
cnodetype type;
union
{
    ⟨node union⟩
};
```

`LAMBDA` defines a function.

Every function expression in the input program is converted to a `LAMBDA` node and associated `cfunction_t`.

Every `LAMBDA` has its own `.lambda.function`, which is a child of the function containing the node. Each element of the `.lambda.closure` array will supply the value of the corresponding variable in `.lambda.function.closure`.

The value which the node produces is a closure, the only kind of user-defined callable object (Section 21.4).

⟨*node union*⟩≡

```
struct
{
    cfunction_t *function;
    cnode_array_t closure;
} lambda;
```

`CALL`/`CALL_FAST` invokes a function.

The node attempts to invoke the value `.call.target` with actual arguments `.args`, yielding the value returned.

`CALL` denotes a "universal" call (Subsection 22.2.2) – the matching of actual to formal arguments will be performed at run-time. `.call.args` may contain `NULL`s, marking omitted arguments. If `.call.names` has nonzero length, each element is either `NULL`, or supplies the name of the formal argument to which the corresponding element in `.call.args` should be assigned.

When this argument matching and type checking has been performed by the compiler, a `CALL_FAST` node stores the result (Subsection 22.2.1). Each bit in `.call.argbits` is set when the corresponding argument is present.

⟨*node union*⟩+≡

```
struct
{
    cnode_t *target;
    cnode_array_t args;
    union
    {
        rsymbol_array_t names;
        argbits_t argbits;
    };
} call;
```

`BUILTIN` generates specialised bytecode.

Some builtin functions (Section 21.6) can be compiled to bytecode instead of being invoked in the usual manner. Semantics and compile-time behaviour are described by the static structure referred to by `.builtin.bi`; its `.ops` callbacks (Section 10.5) will create a `BUILTIN` node from a `CALL` if they deem it desirable. If the instruction has multiple variants, `.builtin.optype` will be used to inform code generation of the types involved in the operation.

⟨*node union*⟩+≡
```
struct
{
    const cbuiltin_t *bi;
    rtype_t *optype;
    cnode_array_t args;
} builtin;
```

`SET`/`BIND` assigns a value to a variable.

An ordinary assignment of `.set.value` to the variable `.set.var` is denoted by a `SET` node.

Introduction of a local variable is performed with a `BIND` to the `.set.value` of its initialising expression. A formal argument is introduced by a `BIND` with `NULL` `.set.value`, in the `.entry` block of the function. The implicit `argbits` argument (Chapter 22) is denoted by a `BIND` with both fields `NULL`.

⟨*node union*⟩+≡
```
struct
{
    cvar_t *var;
    cnode_t *value;
} set;
```

`REF` retrieves a variable's value.

The current value of `.var` is denoted by a `REF` node.

⟨*node union*⟩+≡
```
struct
{
    cvar_t *var;
} ref;
```

`IF` branches conditional on a value.

Control flow will branch to the first element of `.block.succ` when the value of `.ifelse.cond` is `true`, and to the second otherwise. `IF` only occurs as the last node in a block.

⟨*node union*⟩+≡
```
struct
{
    cnode_t *cond;
} ifelse;
```

`RETURN` ends a function.

Control flow within a function is ended, returning to its caller with `.ret.value` as the result of the call. `RETURN` also only occurs last in a block.

⟨*node union*⟩+≡
```
struct
{
    cnode_t *value;
} ret;
```

`COPY` converts a value to another type.

When the compiler needs to ensure a `.value` has some specific type, it will insert a `COPY` node with its `.decl` set appropriately.

This could convert, box, or unbox a scalar. If a reference object is supplied, its type will be checked. Its contents will not be copied or converted.

⟨*node union*⟩+≡
```
struct
{
    cnode_t *value;
} copy;
```

`CONST` yields a constant value.

The `.constant` object is made available as a value for use by other nodes.

⟨*node union*⟩+≡
```
robject_t *constant;
```

`PHI` selects a value conditional on control flow.

The ϕ-function of static single assignment form (Cytron et al., 1991). At the beginning of a block with more than one predecessor, a contiguous sequence of `PHI` nodes may occur. When control flow enters the block from the `i`th predecessor in `.block.pred`, each `PHI` node yields the `i`th element of its `.phi.args`.

⟨*node union*⟩+≡
```
struct
{
    cnode_array_t args;
} phi;
```

5.2 Utilities

Other modules in the compiler use the functions this module provides to manage and manipulate IR structures.

⟨*ir.c*⟩≡
  ⟨*includes*⟩
  ⟨*functions*⟩
  ⟨*blocks*⟩
  ⟨*nodes*⟩
  ⟨*use tracking*⟩
  ⟨*declarations*⟩
  ⟨*variables*⟩

5.2.1 Functions

A `cfunction_t` maps fairly transparently to an `rfunction_t` and, once created created, needs little further modification.

⟨*functions*⟩≡
  ⟨*cfunc_create*⟩
  ⟨*cfunc_free*⟩
  ⟨*cfunc_type*⟩
  ⟨*cfunc_map_children*⟩
  ⟨*cfunc_mapc_children*⟩

If the function has a `parent`, it's added to the latter's list of children.

⟨*cfunc_create*⟩≡

```
cfunction_t *cfunc_create(cfunction_t *parent)
{
    cfunction_t *fn = xcalloc(1, sizeof(*fn));

    fn->parent = parent;
    list_init(&fn->cvar_head);
    list_init(&fn->cfunc_head);
    list_init(&fn->cblock_head);
    array_init(&fn->closure, 0);
    if(parent)
        list_add(&parent->cfunc_head, &fn->cfunc_list);
    else
        list_init(&fn->cfunc_list);
    return fn;
}
```

The function owns, so must free, blocks and bound variables; but not child functions (directly, at least.)

⟨*cfunc_free*⟩≡

```
void cfunc_free(cfunction_t *fn)
{
    cblock_t *block, *btmp;
    cvar_t *var, *vtmp;

    if(fn->parent)
        list_remove(&fn->cfunc_list);
    list_foreach_entry_safe(&fn->cblock_head, block, btmp, cblock_list)
        cblock_free(block);
    list_foreach_entry_safe(&fn->cvar_head, var, vtmp, cvar_list)
        cvar_free(var);
    array_fini(&fn->closure);
    xfree(fn);
}
```

The type of object created by a function expression is determined by its return type and the types of its arguments (Section 21.3). These are not recovered during analysis (Section 10.3), and are assumed to be `object`s unless declared otherwise by the user.

`LEXICAL` variables with `.is_arg` set count as arguments (though the latter implies the former, since only the top-level function will have global variables on its `.cvar_head` list.)

⟨*cfunc_type*⟩≡

```
rtype_t *cfunc_type(cfunction_t *fn, rtype_t *ret_type)
{
    int i = 0;
    cvar_t *var;

    list_foreach_entry(&fn->cvar_head, var, cvar_list)
        if(!cvar_is_global(var) && var->local.is_arg)
            i++;

    argdesc_t *args = alloca(sizeof(argdesc_t) * i);

    i = 0;
    list_foreach_entry(&fn->cvar_head, var, cvar_list)
    {
        if(cvar_is_global(var) || !var->local.is_arg)
            continue;
        args[i++] = argdesc_init(var->name, decl_type(var->decl),
                                 var->local.is_optional);
    }
    return rcall_type_create(decl_type(ret_type), i, args);
}
```

Closure analysis (Section 7.6) will add free variables to a function's `.closure`. If it doesn't find any, the function won't need a closure allocated at run-time, and may be amenable to further optimisation.

⟨*cfunc_has_closure*⟩≡

```
static inline bool cfunc_has_closure(cfunction_t *fn)
    { return alen(&fn->closure) > 0; }
```

Since the program is a tree of `cfunction_t`s, to visit it in its entirety a compilation pass `func` will recursively descend to the children of function `fn`. To simplify this invocation pattern, `cfunc_map_children` accumulates and returns the `cresult` flags (Section 3.2)...

⟨*cfunc_map_children*⟩≡

```
cresult cfunc_map_children(cfunction_t *fn, cresult (*func)(cfunction_t *))
{
    cfunction_t *child;
    cresult res = SUCCESS;

    list_foreach_entry(&fn->cfunc_head, child, cfunc_list)
        res |= func(child);
    return res;
}
```

... and `cfunc_mapc_children` does not.

⟨*cfunc_mapc_children*⟩≡
```
void cfunc_mapc_children(cfunction_t *fn, void (*func)(cfunction_t *))
{
    cfunction_t *child;

    list_foreach_entry(&fn->cfunc_head, child, cfunc_list)
        func(child);
}
```

5.2.2 Variables

Only certain types of variables remain relevant beyond SSA conversion; most code dealing with them is specialised, and resides in other modules.

⟨*variables*⟩≡
  ⟨*cvar_create*⟩
  ⟨*cvar_free*⟩
  ⟨*index_of_var*⟩

It's the caller's responsibility to add the variable to its binder's list, and initialise the correct branch of the union depending on `type`.

⟨*cvar_create*⟩≡
```
cvar_t *cvar_create(rsymbol_t *name, cvartype type, rtype_t *decl)
{
    cvar_t *var = xcalloc(1, sizeof(*var));

    *var = (cvar_t) {
        .name = name,
        .decl = decl,
        .type = type,
    };
    list_init(&var->cvar_list);
    return var;
}
```

Variables own no unmanaged memory. They don't keep backpointers, so it's not safe to `cvar_free` one should anything still refer to it.

⟨*cvar_free*⟩≡
```
void cvar_free(cvar_t *var)
{
    list_remove(&var->cvar_list);
    xfree(var);
}
```

`index_of_var` returns the index at which the element `var` is found in `arr` (or `-1` if it isn't – caveat caller.)

⟨*index_of_var*⟩≡
```
int index_of_var(cvar_array_t *arr, cvar_t *var)
{
    int i;

    array_foreach(arr, i)
    {
        cvar_t *chk = aref(arr, i);
        if(chk == var)
            return i;
    }
    return -1;
}
```

Global variables behave quite differently in some circumstances (they can't be closed over by a `LAMBDA`, for example,) and may be distinguished with the `cvar_is_global` convenience predicate.

⟨*cvar_is_global*⟩≡
```
static inline bool cvar_is_global(cvar_t *var)
    { return var->type != LEXICAL; }
```

A closed variable which may be assigned to (i.e. is not constant) can't be stored as-is in a closure, since we use "flat" closures. The predicate `cvar_is_celled` returns `true` for these; they'll be treated by a cell introduction pass (Chapter 12).

⟨*cvar_is_celled*⟩≡
```
static inline bool cvar_is_celled(cvar_t *var)
    { return !cvar_is_global(var) && !var->is_const && var->local.is_closed; }
```

5.2.3 Blocks

Blocks contain (arrays of) mutual pointers to other blocks, which must be kept synchronised for safe traversal.

⟨*blocks*⟩≡
⟨*cblock_create*⟩
⟨*cblock_free*⟩
⟨*index_of_block*⟩
⟨*link_blocks*⟩
⟨*replace_link*⟩
⟨*remove_link*⟩
⟨*split_block*⟩

After creation, the caller will link the `cblock_t` into the control-flow graph at the appropriate point with `link_blocks`.

⟨*cblock_create*⟩≡
```
cblock_t *cblock_create()
{
    cblock_t *block = xcalloc(1, sizeof(*block));

    array_init(&block->pred, 0);
    array_init(&block->succ, 0);
    list_init(&block->cnode_head);
    return block;
}
```

In general, `_free` functions are not sufficient to safely remove objects. The `.succ` and `.pred` arrays of others which may contain `block` are not updated by `cblock_free`.

⟨*cblock_free*⟩≡
```
void cblock_free(cblock_t *block)
{
    cnode_t *node, *tmp;

    list_remove(&block->cblock_list);
    list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
        cnode_free(node);
    array_fini(&block->pred);
    array_fini(&block->succ);
    xfree(block);
}
```

`link_blocks` adds the edge `from` → `to` to the graph.

⟨*link_blocks*⟩≡
```
void link_blocks(cblock_t *to, cblock_t *from)
{
    array_push(&from->succ, to);
    array_push(&to->pred, from);
}
```

`index_of` for arrays of blocks – useful during `PHI` argument manipulation.

⟨*index_of_block*⟩≡
```
int index_of_block(cblock_array_t *arr, cblock_t *block)
{
    int i;
    array_foreach(arr, i)
    {
        cblock_t *chk = aref(arr, i);
        if(chk == block)
            return i;
    }
    return -1;
}
```

`remove_link` removes the first element matching `from` from `arr`…

⟨*remove_link*⟩≡
```
void remove_link(cblock_array_t *arr, cblock_t *from)
{
    int i = index_of_block(arr, from);
    assert(i != -1);
    array_remove(arr, i);
}
```

…and `replace_link` replaces it with `to`.

⟨*replace_link*⟩≡
```
void replace_link(cblock_array_t *arr, cblock_t *from, cblock_t *to)
{
    int i = index_of_block(arr, from);
    assert(i != -1);
    aset(arr, i, to);
}
```

`split_block` splits the block in-place just above `site`, returning the upper part (splitting on a `PHI` shouldn't be done, as it will break an invariant.)

A new block is created for this. It replaces `site.block` in the `.succ` arrays of the former's predecessors, and is given its `.pred` array and all its nodes prior to the split point.

If it happens to be `fn`'s entry block that's being split, `.entry` is updated appropriately.

⟨*split_block*⟩≡

```
cblock_t *split_block(cfunction_t *fn, cnode_t *site)
{
    cblock_t *pred, *block = site->block, *above = cblock_create();
    cnode_t *node, *tmp;

    assert(site->type != CN_PHI);
    array_foreach_entry(&block->pred, pred)
        replace_link(&pred->succ, block, above);
    above->pred = block->pred;
    array_init(&block->pred, 0);
    list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
    {
        if(node == site)
            break;
        list_remove(&node->cnode_list);
        node->block = above;
        list_add_before(&above->cnode_head, &node->cnode_list);
    }
    list_add_before(&fn->cblock_head, &above->cblock_list);
    if(fn->entry == block)
        fn->entry = above;
    return above;
}
```

5.2.4 Nodes

Nodes are the most complex kind of IR object, and nearly all passes interact with them in some way. They also contain mutual pointers, which must be managed carefully.

⟨*nodes*⟩≡

⟨*cnode_create*⟩
⟨*cnode_append*⟩
⟨*cnode_prepend*⟩
⟨*cnode_insert_before*⟩
⟨*cnode_insert_after*⟩
⟨*cnode_fini*⟩
⟨*cnode_free*⟩
⟨*cnode_reset*⟩
⟨*cnode_remove*⟩
⟨*call_is_pure*⟩
⟨*cnode_is_pure*⟩
⟨*index_of_node*⟩

The caller of `cnode_create` invariably knows what `type` of node it requires, and the `block` to which it will be added.

⟨*cnode_create*⟩≡
```
cnode_t *cnode_create(cblock_t *block, cnodetype type)
{
    cnode_t *node = xcalloc(1, sizeof(*node));

    *node = (cnode_t) {
        .block = block,
        .type = type
    };
    list_init(&node->cnode_list);
    array_init(&node->users, 0);
    return node;
}
```

The block's list of nodes is kept in control-flow order, and there are several possibilities when it comes to extending it with a freshly created `node`: append to the end of the `block`, ...

⟨*cnode_append*⟩≡
```
cnode_t *cnode_append(cblock_t *block, cnodetype type)
{
    cnode_t *node = cnode_create(block, type);
    list_add_before(&block->cnode_head, &node->cnode_list);
    return node;
}
```

... prepend to the beginning, ...

⟨*cnode_prepend*⟩≡
```
cnode_t *cnode_prepend(cblock_t *block, cnodetype type)
{
    cnode_t *node = cnode_create(block, type);
    list_add(&block->cnode_head, &node->cnode_list);
    return node;
}
```

... insert it before some `other` node, ...

⟨*cnode_insert_before*⟩≡
```
cnode_t *cnode_insert_before(cnode_t *other, cnodetype type)
{
    cnode_t *node = cnode_create(other->block, type);
    list_add_before(&other->cnode_list, &node->cnode_list);
    return node;
}
```

... or insert it after.

⟨*cnode_insert_after*⟩≡
```
cnode_t *cnode_insert_after(cnode_t *other, cnodetype type)
{
    cnode_t *node = cnode_create(other->block, type);
    list_add(&other->cnode_list, &node->cnode_list);
    return node;
}
```

The usual `_free` function has been decomposed. `cnode_fini` merely deallocates any memory owned by the `node`.

Since a `LAMBDA` owns its (unique) `.function`, this is the only caller of `cfunc_free`. A `CONST`'s object is interned, and may be shared. Freeing it is left to the garbage collector.

⟨*cnode_fini*⟩≡
```
void cnode_fini(cnode_t *node)
{
    switch(node->type)
    {
    case CN_CALL:
        array_fini(&node->call.names);
        /* fallthrough */
    case CN_CALL_FAST:
        array_fini(&node->call.args);
        break;
    case CN_PHI:
        array_fini(&node->phi.args);
        break;
    case CN_LAMBDA:
        cfunc_free(node->lambda.function);
        array_fini(&node->lambda.closure);
        break;
    case CN_BUILTIN:
        array_fini(&node->builtin.args);
        break;
    default:
        break;
    }
}
```

`cnode_free`, in addition, removes the node from its block's list, deallocates the array of `.users`, and frees the structure itself.

⟨*cnode_free*⟩≡
```
void cnode_free(cnode_t *node)
{
    cnode_fini(node);
    list_remove(&node->cnode_list);
    array_fini(&node->users);
    xfree(node);
}
```

Neither `cnode_fini` nor `cnode_free` should be called on a node which has registered itself as one of another node's `.users`.

`cnode_reset` may be called instead of the former, prior to reusing the `node` in question in-place as another `.type`...

⟨*cnode_reset*⟩≡
```
void cnode_reset(cnode_t *node)
{
    node->decl = NULL;
    cnode_unuse_all(node);
    cnode_fini(node);
}
```

...and `cnode_remove` instead of the latter, to remove it from other node's `.users`, control flow, and memory.

⟨*cnode_remove*⟩≡
```
void cnode_remove(cnode_t *node)
{
    assert(alen(&node->users) == 0);
    cnode_unuse_all(node);
    cnode_free(node);
}
```

The `index_of` a node in an array is also useful when manipulating `PHI` arguments.

⟨*index_of_node*⟩≡
```
int index_of_node(cnode_array_t *arr, cnode_t *node)
{
    int i;

    array_foreach(arr, i)
    {
        cnode_t *chk = aref(arr, i);
        if(chk == node)
            return i;
    }
    return -1;
}
```

A *pure* `node` has no user-observable side effects and, given the same inputs, always produces the same output. `cnode_is_pure` is necessarily conservative, returning `true` only when it's certain to be the case.

When `may_alias` is `false` it's guaranteed that no aliases of the node's result value will be created as a consequence of the call to the predicate; so the latter constraint is relaxed.

⟨*cnode_is_pure*⟩≡
```
bool cnode_is_pure(cnode_t *node, bool may_alias)
{
    switch(node->type)
    {
    case CN_LAMBDA:
    case CN_CONST:
    case CN_REF:
    case CN_PHI:
    case CN_COPY:
        return true;
    case CN_CALL:
    case CN_CALL_FAST:
        return call_is_pure(node, may_alias);
    case CN_BUILTIN:
        return builtin_is_pure(node->builtin.bi, node, may_alias);
    default:
        return false;
    }
}
```

A builtin (Section 21.6) can provide a more detailed account of its functional purity at a particular `node` – either a `BUILTIN`, or a `CALL` of the `CONST` builtin itself. In the latter case, the predicate is called on the extracted value by the `call_is_pure` helper.

⟨*call_is_pure*⟩≡
```
static inline bool call_is_pure(cnode_t *node, bool may_alias)
{
    if(node->call.target->type == CN_CONST)
    {
        robject_t *val = node->call.target->constant;

        if(rtype_is_callable(r_typeof(val))
           && rcall_is_builtin((rcallable_t *)val))
        {
            const cbuiltin_t *bi = ((rbuiltin_t *)val)->cbi;

            return bi && builtin_is_pure(bi, node, may_alias);
        }
    }
    return false;
}
```

`IF`, `RETURN` and `SET` nodes do not yield values, and should be referred to by no other nodes. Some `BUILTIN`s don't, either, depending on their semantics as described by `.bi` (assuming it's present – when IR is being built, it may not be.)

⟨*cnode_yields_value*⟩≡
```
static inline bool cnode_yields_value(cnode_t *node)
{
    switch(node->type)
    {
    case CN_IF:
    case CN_RETURN:
    case CN_SET:
        return false;
    case CN_BUILTIN:
        if(node->builtin.bi)
            return !builtin_is_void(node->builtin.bi, node);
        /* fallthrough */
    default:
        return true;
    }
}
```

When a `CALL` has named actual arguments, they're added to its `.names` array.

⟨*call_has_names*⟩≡
```
static inline bool call_has_names(cnode_t *node)
    { return alen(&node->call.names) > 0; }
```

5.2.5 Use Tracking

A node's union fields contain, according to its `.type`, pointers to the nodes that it uses to determine its value. Each of the latter also keeps a backpointer to the former, so we can efficiently follow value flow in the other direction.

⟨*use tracking*⟩≡
  ⟨*cnode_map_used*⟩
  ⟨*cnode_add_user*⟩
  ⟨*cnode_remove_user*⟩
  ⟨*add_one*⟩
  ⟨*cfunc_node_users*⟩
  ⟨*remove_one*⟩
  ⟨*cnode_unuse_all*⟩
  ⟨*replace_ctx_t*⟩
  ⟨*replace_one*⟩
  ⟨*cnode_replace_in_users*⟩

`cnode_map_used` invokes the callback `func` on (a pointer to) each input used by the given node. An optional pointer to context `data` is also passed along.

`NULL`s can be present in a `CALL`/`CALL_FAST`/`BUILTIN` argument list or a `LAMBDA` closure, and are ignored if found.

⟨*cnode_map_used*⟩≡
```
#define NODE(f) func(&node->f, data)
#define NODES(f)                              \
    array_foreach_ptr(&node->f, ptr) {        \
        if(*ptr)                              \
            func(ptr, data);                  \
    }
void cnode_map_used(cnode_t *node, void (*func)(cnode_t **, void *),
                    void *data)
{
    cnode_t **ptr;

    switch(node->type)
    {
    case CN_IF: NODE(ifelse.cond); break;
    case CN_RETURN: NODE(ret.value); break;
    case CN_COPY: NODE(copy.value); break;
    case CN_CALL:
    case CN_CALL_FAST:
        NODE(call.target);
        NODES(call.args);
        break;
    case CN_BIND:
        if(!node->set.value)
            break;
        /* fallthrough */
    case CN_SET: NODE(set.value); break;
    case CN_PHI: NODES(phi.args); break;
    case CN_BUILTIN: NODES(builtin.args); break;
    case CN_LAMBDA: NODES(lambda.closure); break;
    default: break;
    }
}
```

A backpointer to `user` can be added to the `.users` array of `node` with `cnode_add_user`. Duplicate uses are not treated specially.

⟨*cnode_add_user*⟩≡
```
void cnode_add_user(cnode_t *user, cnode_t *node)
{
    array_push(&node->users, user);
}
```

`cnode_remove_user` is the inverse, removing one use.

⟨*cnode_remove_user*⟩≡
```
void cnode_remove_user(cnode_t *user, cnode_t *node)
{
    int i;

    array_foreach(&node->users, i)
    {
        if(aref(&node->users, i) == user)
        {
            array_remove(&node->users, i);
            return;
        }
    }
}
```

`cnode_is_used` returns `true` if `node`'s value is used by any other...

⟨*cnode_is_used*⟩≡
```
static inline bool cnode_is_used(cnode_t *node)
    { return !array_isempty(&node->users); }
```

...and `cnode_only_used_by` returns `true` if `node` is only used by `user`.

⟨*cnode_only_used_by*⟩≡
```
static inline bool cnode_only_used_by(cnode_t *node, cnode_t *user)
{
    cnode_t *chk;

    if(!cnode_is_used(node))
        return false;
    array_foreach_entry(&node->users, chk)
        if(chk != user)
            return false;
    return true;
}
```

`cfunc_node_users` initialises the `.users` array, which is fairly simple with the machinery in place. Visiting each `node` in each `block` in the function, the `add_one` callback...

⟨*cfunc_node_users*⟩≡
```
void cfunc_node_users(cfunction_t *fn)
{
    cblock_t *block;
    cnode_t *node;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
        list_foreach_entry(&block->cnode_head, node, cnode_list)
            cnode_map_used(node, add_one, node);
}
```

...registers the `node` as a user of each of its inputs.

⟨*add_one*⟩≡
```
static void add_one(cnode_t **ptr, void *data)
    { cnode_add_user(data, *ptr); }
```

And `cnode_unuse_all`, called on `_remove` or `_reset`, unregisters the `node`...

⟨*cnode_unuse_all*⟩≡
```
void cnode_unuse_all(cnode_t *node)
    { cnode_map_used(node, remove_one, node); }
```

...with the `remove_one` callback.

⟨*remove_one*⟩≡
```
static void remove_one(cnode_t **ptr, void *data)
    { cnode_remove_user(data, *ptr); }
```

5.2.6 Replacement

`cnode_map_used` also lets us replace one node with another (that computes an equivalent value.)

⟨*replace_ctx_t*⟩≡
```
typedef struct
{
    cnode_t *from, *to, *node;
} replace_ctx_t;
```

`cnode_replace_in_users` implements this functionality. It calls `replace_one` for each input of each `node` that uses `from`; then the array of `.users` is emptied. It's now safe for the caller to call `cnode_remove`.

⟨*cnode_replace_in_users*⟩≡
```
void cnode_replace_in_users(cnode_t *from, cnode_t *to)
{
    replace_ctx_t ctx = { from, to };
    cnode_t *node;

    array_foreach_entry(&from->users, node)
    {
        ctx.node = node;
        cnode_map_used(node, replace_one, &ctx);
    }
    array_resize(&from->users, 0);
}
```

If the input pointer matches `.from`, it's replaced with `.to`, and the fact noted in the latter's `.users`.

⟨*replace_one*⟩≡
```
static void replace_one(cnode_t **ptr, void *data)
{
    replace_ctx_t *ctx = data;

    if(*ptr == ctx->from)
    {
        cnode_add_user(ctx->node, ctx->to);
        *ptr = ctx->to;
    }
}
```

5.2.7 Type Declarations

The language is not statically typed. However, the compiler can use the optionally declared types of variables to determine upper bounds for the types of some values, then propagate these bounds through the input program to recover the types of others.

```
⟨declarations⟩≡
  ⟨nil_init⟩
  ⟨guard_decl⟩
  ⟨guard_val⟩
  ⟨enforce_decl⟩
  ⟨enforce_val⟩
```

An undeclared variable can contain values of any type. Its `.decl` is `NULL`, and its type is considered to be `object`.

```
⟨decl_type⟩≡
  static inline rtype_t *decl_type(rtype_t *decl)
      { return decl ? decl : r_type_object; }
```

`decl_name` is convenient when formatting diagnostic messages which mention declarations.

```
⟨decl_name⟩≡
  static inline const char *decl_name(rtype_t *decl)
      { return rtype_name(decl_type(decl)); }
```

According to these conventions, `cnode_compat` tests if `node`'s currently declared or recovered type `.decl` is compatible (Section 20.5) with the given type `decl`.

```
⟨cnode_compat⟩≡
  static inline compat cnode_compat(cnode_t *node, rtype_t *decl)
      { return r_type_compat(decl_type(node->decl), decl_type(decl), true); }
```

A variable which isn't explicitly initialised is given a zero value. The type of the latter must agree with the declared type of the former, if any.

```
⟨nil_init⟩≡
  robject_t *nil_init(rtype_t *decl)
  {
      robject_t *obj = NULL;

      if(decl && rtype_is_scalar(decl))
      {
          rvalue_union_t val = { 0 };
          obj = r_box(decl, &val);
      }
      return obj;
  }
```

Declarations are treated as *assertions* (MacLachlan, 1992), so checks or conversions must be inserted where the types of values provided could differ from those which are expected.

A `COPY` node may be used to *guard* another node's value. The resulting bytecode will convert or check `val` against the type `decl`. If control flow continues past the guard, its value will be of the correct type.

To guard a node's input value, the `node` no longer uses `val` directly; it uses the `guard` instead, which uses the value on its behalf.

⟨*guard_decl*⟩≡

```
cnode_t *guard_decl(cnode_t *guard, cnode_t *node, rtype_t *decl,
                    cnode_t *val)
{
    guard->copy.value = val;
    guard->decl = decl_type(decl);
    guard->file = val->file;
    cnode_remove_user(node, val);
    cnode_add_user(node, guard);
    cnode_add_user(guard, val);
    return guard;
}
```

The result value of a node can be guarded similarly. All users of the `node` are made to use the `guard` instead.

⟨*guard_val*⟩≡

```
static cnode_t *guard_val(cnode_t *guard, cnode_t *node, rtype_t *decl)
{
    guard->copy.value = node;
    guard->decl = decl_type(decl);
    guard->file = node->file;
    cnode_replace_in_users(node, guard);
    cnode_add_user(guard, node);
    return guard;
}
```

`enforce_decl` ensures that the input value `*pval` of the `node` is of the specific type `decl`.

If the type of the value is a subtype of the last, it's already correct. If not, any run-time check at this point would always fail, so the error is signalled.

If it can't be statically determined (perhaps the input is only known to be an `object`,) a guard is inserted before the node.

`boxing` is `false` when it's safe to ignore the scalar calling convention (which requires that scalars are boxed when passed to functions expecting reference objects.)

⟨*enforce_decl*⟩≡

```
cresult enforce_decl(cnode_t *node, rtype_t *decl, cnode_t **pval,
                     bool boxing)
{
    switch(cnode_compat(*pval, decl))
    {
    case YES:
        break;
    case NO:
        c_error("value of type ‘%s‘ found where type ‘%s‘ expected.",
                decl_name((*pval)->decl), decl_name(decl));
        return FAILED;
    case MAYBE:
        if(!boxing && decl_type(decl) == r_type_object)
```

```
            break;
        *pval = guard_decl(cnode_insert_before(node, CN_COPY),
                           node, decl, *pval);
        return CHANGED;
    }
    return SUCCESS;
}
```

Through recovery, the compiler may discover the precise return type of a `CALL` to a builtin given the types of its arguments and, if it does, will set its `.decl` accordingly.

However, the runtime will call it according to its signature, which is more general and specifies a return type of `decl` instead. `enforce_val` hides this (ugly) implementation detail from the users of `node` by guarding its result value if necessary.

⟨*enforce_val*⟩≡

```
cresult enforce_val(cnode_t *node, rtype_t *decl)
{
    switch(cnode_compat(node, decl))
    {
    case YES:
        break;
    case NO:
        c_error("value of type ‘%s‘ found where type ‘%s‘ expected.",
                decl_name(node->decl), decl_name(decl));
        return FAILED;
    case MAYBE:
        guard_val(cnode_insert_after(node, CN_COPY), node, node->decl);
        node->decl = decl;
        return CHANGED;
    }
    return SUCCESS;
}
```

Miscellanea

⟨*preliminaries*⟩≡

```
typedef struct cnode cnode_t;
typedef struct cblock cblock_t;
typedef struct cfunction cfunction_t;
typedef struct cvar cvar_t;
typedef ARRAY(cfunction_t *) cfunction_array_t;
typedef ARRAY(cblock_t *) cblock_array_t;
typedef ARRAY(cnode_t *) cnode_array_t;
typedef ARRAY(cvar_t *) cvar_array_t;
typedef ARRAY(rsymbol_t *) rsymbol_array_t;
```

⟨*includes*⟩≡

```
#include "global.h"
#include "ir.h"
```

⟨*prototypes*⟩≡
```
cfunction_t *cfunc_create(cfunction_t *parent);
void cfunc_free(cfunction_t *fn);
rtype_t *cfunc_type(cfunction_t *fn, rtype_t *ret_type);
cresult cfunc_map_children(cfunction_t *fn, cresult (*func)(cfunction_t *));
void cfunc_mapc_children(cfunction_t *fn, void (*func)(cfunction_t *));

cblock_t *cblock_create();
void cblock_free(cblock_t *block);
int index_of_block(cblock_array_t *arr, cblock_t *block);
void link_blocks(cblock_t *to, cblock_t *from);
void replace_link(cblock_array_t *arr, cblock_t *from, cblock_t *to);
void remove_link(cblock_array_t *arr, cblock_t *from);
cblock_t *split_block(cfunction_t *fn, cnode_t *site);

cnode_t *cnode_create(cblock_t *block, cnodetype type);
cnode_t *cnode_prepend(cblock_t *block, cnodetype type);
cnode_t *cnode_append(cblock_t *block, cnodetype type);
cnode_t *cnode_insert_before(cnode_t *other, cnodetype type);
cnode_t *cnode_insert_after(cnode_t *other, cnodetype type);
int index_of_node(cnode_array_t *arr, cnode_t *node);
void cnode_fini(cnode_t *node);
void cnode_free(cnode_t *node);
void cnode_reset(cnode_t *node);
void cnode_remove(cnode_t *node);

void cnode_unuse_all(cnode_t *node);
void cnode_replace_in_users(cnode_t *from, cnode_t *to);
void cnode_add_user(cnode_t *user, cnode_t *node);
void cnode_remove_user(cnode_t *user, cnode_t *node);
void cnode_map_used(cnode_t *node, void (*func)(cnode_t **, void *), void *data);
void cfunc_node_users(cfunction_t *fn);

bool cnode_is_pure(cnode_t *node, bool may_alias);

cnode_t *guard_decl(cnode_t *guard, cnode_t *node, rtype_t *decl, cnode_t *val);
cresult enforce_decl(cnode_t *node, rtype_t *decl, cnode_t **pval, bool boxing);
cresult enforce_val(cnode_t *node, rtype_t *decl);
robject_t *nil_init(rtype_t *decl);

cvar_t *cvar_create(rsymbol_t *name, cvartype type, rtype_t *decl);
void cvar_free(cvar_t *var);
int index_of_var(cvar_array_t *arr, cvar_t *block);

cresult cfunc_crit_edges(cfunction_t *fn); // FIXME can remove from here when we stop asserti
void cfunc_rdfo(cfunction_t *fn);
void cfunc_cleanup(cfunction_t *fn);
void cfunc_init_closure(cfunction_t *fn);
void cfunc_ssa_convert(cfunction_t *fn);

#include "ir_dom.h"

// XXX really somewhere in opt, because builtin_ops_t is private
bool builtin_is_void(const cbuiltin_t *bi, cnode_t *node);
bool builtin_is_pure(const cbuiltin_t *bi, cnode_t *node, bool may_alias);
void ir_dump(cfunction_t *fn); // DEBUG
```

⟨*inlines*⟩≡
  ⟨*cfunc_has_closure*⟩
  ⟨*cnode_is_used*⟩
  ⟨*cnode_yields_value*⟩
  ⟨*call_has_names*⟩
  ⟨*cnode_only_used_by*⟩
  ⟨*decl_type*⟩
  ⟨*decl_name*⟩
  ⟨*cnode_compat*⟩
  ⟨*cvar_is_global*⟩
  ⟨*cvar_is_celled*⟩

Chapter 6

AST Conversion

Given an abstract syntax tree parsed from a user program (Section 4.1), *conversion* creates from it the functions, variables, blocks and nodes (Chapter 5) upon which the rest of the compiler will operate.

⟨*ir_convert.c*⟩≡
  ⟨*includes*⟩
  ⟨*diagnostics*⟩
  ⟨*environments*⟩
  ⟨*building*⟩
  ⟨*bindings*⟩
  ⟨*forms*⟩
  ⟨*conversion*⟩
  ⟨*check_resolve*⟩
  ⟨*resolve_global*⟩
  ⟨*ir_convert*⟩

6.1 Environments

Lexically nested constructs give rise to linked *environments*.

⟨*environments*⟩≡
  ⟨*function env*⟩
  ⟨*lexical env*⟩
  ⟨*loop env*⟩

6.1.1 Function Environment

The function environment `funenv_t` tracks the function (Subsection 5.1.1)currently being converted. . .

⟨*function env*⟩≡
  ⟨*funenv_t*⟩
  ⟨*fun_open*⟩
  ⟨*fun_add_block*⟩
  ⟨*fun_close*⟩

. . . in the `.function` field. The `.cblock_head` list contains the basic blocks (Subsection 5.1.3) which will form its body, in no particular order.

⟨*funenv_t*⟩≡
```
typedef struct funenv
{
    cfunction_t *function;
    list_t cblock_head;
} funenv_t;
```

`fun_open` initialises the environment `fun`, lexically nested inside its `parent` (which is `NULL` if the former is the *top-level function*, created in Section 6.3.)

⟨*fun_open*⟩≡
```
static funenv_t *fun_open(funenv_t *fun, funenv_t *parent)
{
    fun->function = cfunc_create(parent ? parent->function : NULL);
    list_init(&fun->cblock_head);
    return fun;
}
```

A new block is created, added to the function, and returned by `fun_add_block`.

⟨*fun_add_block*⟩≡
```
static cblock_t *fun_add_block(funenv_t *fun)
{
    cblock_t *block = cblock_create();
    list_add(&fun->cblock_head, &block->cblock_list);
    return block;
}
```

When conversion of a function's body is complete, `fun_close` initialises the `.function`'s `entry` block and computes the type of the closures that it will create, taking arguments and return type `ret_type` into account. The resulting `cfunction_t` is returned.

At this point the blocks are still in the environment's `.cblock_head` list. Starting at `entry`, `cfunc_rdfo` traverses the control-flow graph, relinking blocks into `fn`'s list in reverse-depth-first order (Section 7.5).

⟨*fun_close*⟩≡
```
static cfunction_t *fun_close(funenv_t *fun, cblock_t *entry,
                              rtype_t *ret_type)
{
    cfunction_t *fn = fun->function;

    fn->entry = entry;
    fn->cl_type = cfunc_type(fn, ret_type);
    cfunc_rdfo(fn);
    return fn;
}
```

Unreachable blocks should not be created in the first place (it would be nice to verify this, but `assert`s have sufficed so far.)

6.1.2 Lexical Environment

A named lexical variable is *bound* to a value by some syntactic construct. The variable refers to that value only within the *scope* of the construct.

⟨*lexical env*⟩≡
  ⟨*lexenv_t*⟩
  ⟨*env_add*⟩
  ⟨*env_add_global*⟩
  ⟨*env_find*⟩
  ⟨*env_find_add*⟩
  ⟨*env_open*⟩
  ⟨*env_close*⟩

A `lexenv_t` is associated with each scope during conversion. `.parent` points outward to the enclosing environment. `.vars` maps `rsymbol_t` names to `cvar_t` variables; it's lazily created, and initially `NULL`. The distinct, shared `.global` environment is outside all others.

⟨*lexenv_t*⟩≡
```
typedef struct lexenv lexenv_t;
typedef struct lexenv
{
    hashmap_t *vars;
    lexenv_t *parent;
    lexenv_t *global;
} lexenv_t;
```

`env_open` enters an environment `env` for a new scope within `parent`. If the latter is `NULL`, it's the outermost, so must be the `.global` itself.

⟨*env_open*⟩≡
```
static lexenv_t *env_open(lexenv_t *env, lexenv_t *parent)
{
    *env = (lexenv_t) {
        .vars = NULL,
        .parent = parent,
        .global = parent ? parent->global : env,
    };
    return env;
}
```

`.vars` is freed by `env_close` when leaving the scope.

⟨*env_close*⟩≡
```
static void env_close(lexenv_t *env)
{
    if(env->vars)
        hashmap_free(env->vars);
}
```

`env_find` recursively searches for a variable of given `name` in the `.vars` of each `lexenv_t`, in the list starting at `env` and proceeding by `.parent`. `NULL` is returned on failure.

⟨*env_find*⟩≡
```
static cvar_t *env_find(lexenv_t *env, rsymbol_t *name)
{
    bool found = false;
    cvar_t *var;

    if(env->vars)
    {
        var = hashmap_get(env->vars, name, &found);
        if(found)
            return var;
    }
    if(env->parent)
        return env_find(env->parent, name);
    return NULL;
}
```

`env_add_global` creates a new variable, adding it to the `.global` environment. Initially of `.type` `GLOBAL_EXT`, it could be declared later in the input, or have its `.extl.global` field resolved to some `rglobal_t` at conversion's end.

⟨*env_add_global*⟩≡
```
static cvar_t *env_add_global(lexenv_t *env, rsymbol_t *name)
{
    cvar_t *var = cvar_create(name, GLOBAL_EXT, NULL);

    var->extl.global = NULL;
    env_add(env->global, var);
    return var;
}
```

`env_add` extends `env` with `var`. Variables of the same name in enclosing environments are *shadowed*; together with `env_find`, this accomplishes α-renaming. Within a single environment, a later binding will replace an earlier one. `.vars` is created if the environment is currently empty.

⟨*env_add*⟩≡
```
static cvar_t *env_add(lexenv_t *env, cvar_t *var)
{
    if(!env->vars)
        env->vars = hashmap_create(rsym_hash, ptr_eq);
    hashmap_set(env->vars, var->name, var);
    return var;
}
```

When `env_find_add` is called, if no variable of that `name` can be found, one is created and added to the global environment.

⟨*env_find_add*⟩≡
```
static cvar_t *env_find_add(lexenv_t *env, rsymbol_t *name)
{
    cvar_t *var = env_find(env, name);

    if(var)
        return var;
    return env_add_global(env, name);
}
```

6.1.3 Loop Environment

The loop environment keeps track of the blocks to which the "`break`" and "`continue`" keywords will branch – these being the `.brk` and `.cont` fields, respectively.

⟨*loop env*⟩≡
```
typedef struct
{
    cblock_t *brk, *cont;
} loopenv_t;
```

6.2 Building

A *builder* structure represents the context in which conversion takes place, appending nodes (Subsection 5.1.4) to the function under construction.

⟨*building*⟩≡
```
⟨ir_builder_t⟩
⟨prototypes⟩
⟨is_top_level⟩
⟨ir_open⟩
⟨ir_build⟩
⟨ir_extend⟩
⟨build_constant⟩
⟨build_ref⟩
⟨build_call⟩
⟨build_phi⟩
⟨build_if⟩
⟨build_set⟩
⟨build_return⟩
⟨build_bind⟩
⟨build_lambda⟩
⟨build_builtin⟩
⟨constant_from_literal⟩
```

6.2.1 IR Builder

An `ir_builder_t` simplifies construction of control and value flow. The `.head` and `.tail` fields track the single entry and exit blocks in the connected subgraph. If the last node built computes a `.value`, that is also tracked.

`.fun`, `.env` and `.loop` point to the environments currently in effect. When `.pfailed` is non-`NULL`, it points to a flag which will be set if an error should be encountered during conversion.

⟨*ir_builder_t*⟩≡
```
typedef struct ir_builder
{
    rsymbol_t *file;
    cblock_t *head, *tail;
    bool *pfailed;
    cnode_t *value;
    funenv_t *fun;
    lexenv_t *env;
    loopenv_t *loop;
} ir_builder_t;
```

The `ir_error` macro sets this flag, and emits the given message as an `ERROR`. `ir_warning` does just the latter, and at the `WARNING` level.

⟨*diagnostics*⟩≡
```
#define ir_error(ir, fmt, args...) do {                        \
        if(ir->pfailed) *ir->pfailed = true;                   \
        c_message_va(C_ERROR, ir->file, fmt, ##args);          \
    } while (0)
#define ir_warning(ir, fmt, args...)                           \
    c_message_va(C_WARNING, ir->file, fmt, ##args)
```

A convenience predicate returns `true` when the immediately enclosing function is at top level.

⟨*is_top_level*⟩≡
```
static inline bool is_top_level(ir_builder_t *ir)
    { return !ir->fun->function->parent; }
```

Initialised with `ir_open`, a builder `ir` can begin afresh, or share the environments (and `.pfailed`) of `another`. Either way, a new block is added to the function under construction.

⟨*ir_open*⟩≡
```
static ir_builder_t *ir_open(ir_builder_t *ir, ir_builder_t *other)
{
    if(other)
        *ir = *other;
    assert(ir->fun);
    ir->head = ir->tail = fun_add_block(ir->fun);
    ir->value = NULL;
    return ir;
}
```

`ir_build` constructs a node of given `type`. It's appended to the `.tail` of the builder; `.value` is updated if it yields a value (and is not a `BIND`; their values aren't to be used directly, yet.)

⟨*ir_build*⟩≡
```
static cnode_t *ir_build(ir_builder_t *ir, cnodetype type)
{
    cnode_t *node = cnode_append(ir->tail, type);
    node->file = ir->file;
    if(type != CN_BIND && cnode_yields_value(node))
        ir->value = node;
    else
        ir->value = NULL;
    return node;
}
```

`ir_extend` advances `ir` so that further construction may continue from the `.tail` of `other` with the `.value` of the latter.

⟨*ir_extend*⟩≡
```
static ir_builder_t *ir_extend(ir_builder_t *ir, ir_builder_t *other)
{
    ir->tail = other->tail;
    ir->value = other->value;
    return ir;
}
```

6.2.2 Nodes

A set of convenience functions are used to construct the various `.types` of `cnode_t`.

The constant object to which a `CONST` node refers is first interned in the compiler's constant table. The node's type `.decl` is also initialised at this point, as it is unlikely to change.

⟨*build_constant*⟩≡
```
static cnode_t *build_constant(ir_builder_t *ir, robject_t *ptr)
{
    cnode_t *node = ir_build(ir, CN_CONST);
    robject_t *obj = c_intern(ptr);

    node->constant = obj;
    node->decl = r_typeof(obj);
    return node;
}
```

The `.var` field of a `REF` node is initialised to the variable found by `name` in the enclosing environment(s). A free variable is assumed to be global.

⟨*build_ref*⟩≡
```
static cnode_t *build_ref(ir_builder_t *ir, rsymbol_t *name)
{
    cnode_t *node = ir_build(ir, CN_REF);

    node->ref.var = env_find_add(ir->env, name);
    return node;
}
```

The invocation of a `target` begins as a `CALL` node; later optimisation may replace it with an equivalent `CALL_FAST` or `BUILTIN`. `names`, together with `args`, have their ownership transferred to the node.

⟨*build_call*⟩≡
```
static cnode_t *build_call(ir_builder_t *ir, cnode_t *target,
                           cnode_array_t *args, rsymbol_array_t *names)
{
    cnode_t *node = ir_build(ir, CN_CALL);

    node->call.target = target;
    node->call.args = *args;
    node->call.names = *names;
    return node;
}
```

An `IF` node only has the condition `cond` to initialise, as the branches are represented by successor blocks in the control flow.

⟨*build_if*⟩≡
```
static cnode_t *build_if(ir_builder_t *ir, cnode_t *cond)
{
    cnode_t *node = ir_build(ir, CN_IF);

    node->ifelse.cond = cond;
    return node;
}
```

A PHI node, as constructed, joins values from the `left` and `right` edges incoming to a control-flow join.

⟨*build_phi*⟩≡
```
static cnode_t *build_phi(ir_builder_t *ir, cnode_t *left, cnode_t *right)
{
    cnode_t *node = ir_build(ir, CN_PHI);
    cnode_array_t *args = &node->phi.args;

    assert(left && right);
    array_init(args, 2);
    array_push(args, left);
    array_push(args, right);
    return node;
}
```

RETURN can be explicit or implied. In the latter case, `nil` will be returned if no meaningful value is available.

⟨*build_return*⟩≡
```
static cnode_t *build_return(ir_builder_t *ir)
{
    cnode_t *prev = ir->value ? ir->value : build_constant(ir, NULL),
            *node = ir_build(ir, CN_RETURN);

    node->ret.value = prev;
    return node;
}
```

The value of a SET node comes from the right-hand side of the assignment.

⟨*build_set*⟩≡
```
static cnode_t *build_set(ir_builder_t *ir, cvar_t *var, cnode_t *val)
{
    cnode_t *node = ir_build(ir, CN_SET);

    node->set.var = var;
    node->set.value = ir->value = val;
    return node;
}
```

A BIND node introduces the variable `var` with optional initial `val` and type `decl`. To preserve flexibility, it's added to the environment in a separate step.

⟨*build_bind*⟩≡
```
static cnode_t *build_bind(ir_builder_t *ir, cvar_t *var,
                           rtype_t *decl, cnode_t *val)
{
    cnode_t *node = ir_build(ir, CN_BIND);

    node->set.value = val;
    node->set.var = var;
    node->decl = decl;
    return node;
}
```

When a LAMBDA is constructed for a cfunction_t, the .node backpointer can be initialised, as they are in one-to-one relation.

⟨*build_lambda*⟩≡
```
static cnode_t *build_lambda(ir_builder_t *ir, cfunction_t *fn)
{
    cnode_t *node = ir_build(ir, CN_LAMBDA);

    array_init(&node->lambda.closure, 0);
    node->lambda.function = fn;
    fn->node = node;
    return node;
}
```

A BUILTIN node usually results from optimisation, but can also be constructed directly.

⟨*build_builtin*⟩≡
```
static cnode_t *build_builtin(ir_builder_t *ir, const cbuiltin_t *bi,
                              cnode_array_t *args, rtype_t *optype, rtype_t *decl)
{
    cnode_t *node = ir_build(ir, CN_BUILTIN);

    node->decl = decl;
    node->builtin.bi = bi;
    node->builtin.args = *args;
    node->builtin.optype = optype;
    return node;
}
```

6.2.3 Constants

Literals in the input AST are converted to robject_ts referenced by CONST nodes.

INT, DOUBLE and STRING literals become the appropriate type of object. No distinction is made between the various .types of .symbol. A TOKEN is also a symbol, albeit named via ast_str.

⟨*constant_from_literal*⟩≡
```
static robject_t *constant_from_literal(ast_t *ast)
{
    switch(ast->type)
    {
    case AST_INT: return r_box(r_type_int, &ast->integer);
    case AST_DOUBLE: return r_box(r_type_double, &ast->dfloat);
    case AST_STRING: return (robject_t *)rstr_create(ast->string);
    case AST_TOKEN: return (robject_t *)r_intern(ast_str(ast));
    case AST_QUOTED:
    case AST_SYMBOL:
    case AST_NAME:
        return (robject_t *)ast->symbol;
    case AST_INVALID: return NULL;
    default: break;
    }
    return NULL; /* NOTREACHED */
}
```

6.3 Entry Point

The entry point to this module is `ir_convert`. It returns a nullary `cfunction_t`, the body of which contains the IR equivalent of the input `ast`.

⟨*ir_convert*⟩≡
```
cfunction_t *ir_convert(ast_t *ast, char *filename)
{
    cfunction_t *function;
    funenv_t fun;
    lexenv_t global_env, env;
    bool failed = false;
    ⟨convert toplevel⟩
}
```

The global lexical environment `global_env` is distinct from (and the parent of) the lexical environment `env` of the top-level function. This allows distinguishing e.g. "`global x`" from "`let x`".

⟨*convert toplevel*⟩≡
```
env_open(&global_env, NULL);
ir_builder_t ir = {
    .fun = fun_open(&fun, NULL),
    .env = env_open(&env, &global_env),
    .file = filename ? r_intern(filename) : NULL,
    .pfailed = &failed
};
ir_open(&ir, NULL);
```

The input is converted by `convert_file`. If control flow continues through it, a `RETURN` node is appended. Free variables have been collected in `global_env`; resolution picks up declared types and constant values from the runtime environment.

⟨*convert toplevel*⟩+≡
```
if(convert_file(&ir, ast))
    build_return(&ir);
function = fun_close(&fun, ir.head, NULL);
env_close(&env);
if(global_env.vars)
    hashmap_map(global_env.vars, resolve_global, &ir);
env_close(&global_env);
```

`NULL` is returned on conversion failure.

⟨*convert toplevel*⟩+≡
```
if(failed)
{
    cfunc_free(function);
    return NULL;
}
return function;
```

If the `global` corresponding to the free variable `var` exists in the runtime global environment, `check_resolve` copies any reliable information from its declaration and verifies that it wasn't declared more than once.

The `cvar_t` is to share the top-level `cfunction_t`'s lifetime, so is added to the latter's list of variables.

⟨*resolve_global*⟩≡
```
static void resolve_global(const void *key, void *value, void *ptr)
{
    ir_builder_t *ir = ptr;
    cvar_t *var = value;
    rglobal_t *global = r_get_global(var->name);
    cfunction_t *fn = ir->fun->function;

    if(global && !check_resolve(var, global))
        *ir->pfailed = true;

    assert(cvar_is_global(var));
    assert(list_isempty(&var->cvar_list));
    list_add(&fn->cvar_head, &var->cvar_list);
}
```

An existing global (19.5) may not be redeclared. If declared explicitly, it may not be removed – so its type `.decl` and constancy `.is_const` are invariants which will hold for the lifetime of the system, and the compiler can rely on them.

⟨*check_resolve*⟩≡
```
static inline bool check_resolve(cvar_t *var, rglobal_t *global)
{
    if(var->type == GLOBAL_INT)
    {
        c_error("duplicate %s definition ‘%s‘",
                var->is_const ? "const" : "global", r_symstr(var->name));
        return false;
    }
    if(global->decl)
    {
        var->extl.global = global;
        var->decl = global->decl;
        var->is_const = global->is_const;
    }
    return true;
}
```

6.4 Conversion

Conversion proceeds through the recursive destructuring of an `ast_t` by `.type`.

⟨*conversion*⟩≡
⟨*convert_literal*⟩
⟨*convert_symbol*⟩
⟨*convert_token*⟩
⟨*convert_node*⟩
⟨*convert_stmt*⟩
⟨*convert_expr*⟩

A `convert_` function returns `true` when control flow continues through the IR it creates – this is not the case for `BREAK`, `CONTINUE`, `RETURN`, or when an error is encountered.

In addition, some callers expect the conversion of a subtree `ast` to produce a `.value`. The grammar does not enforce this; instead, `convert_expr` returns `true` when the constraint is satisfied, signalling an error otherwise (if one hasn't been, already.)

⟨*convert_expr*⟩≡
```
static bool convert_expr(ir_builder_t *ir, ast_t *ast)
{
    if(convert_stmt(ir, ast) && ir->value)
        return true;
    if(ir->pfailed && !*ir->pfailed)
        ir_error(ir, "statement found where expression expected.");
    return false;
}
```

`convert_stmt` is the fundamental conversion function. `INT`, `DOUBLE`, `QUOTED` and `STRING` literal `ast`s are handled by `convert_literal`. Unquoted `SYMBOL`s, denoting references to variables, are handled by `convert_symbol`. A `NODE` requires further destructuring by `convert_node`. The single tokens `BREAK` and `CONTINUE` become control flow in `convert_loop_exit`; no others are valid.

⟨*convert_stmt*⟩≡
```
static bool convert_stmt(ir_builder_t *ir, ast_t *ast)
{
    switch(ast->type)
    {
    case AST_INT:
    case AST_DOUBLE:
    case AST_QUOTED:
    case AST_STRING:
        return convert_literal(ir, ast);
    case AST_SYMBOL:
        return convert_symbol(ir, ast);
    case AST_NODE:
        return convert_node(ir, ast);
    case AST_TOKEN:
        if(ast->token == BREAK || ast->token == CONTINUE)
            return convert_loop_exit(ir, ast);
        /* fallthrough */
    default:
        ir_error(ir, "invalid AST of type %d", ast->type);
    }
    return false; /* NOTREACHED, ideally */
}
```

For a literal, a `CONST` node is built...

⟨*convert_literal*⟩≡
```
static bool convert_literal(ir_builder_t *ir, ast_t *ast)
{
    build_constant(ir, constant_from_literal(ast));
    return true;
}
```

. . . and, for a variable reference, a `REF` node (although "`...`" is not valid at the top level.)

⟨*convert_symbol*⟩≡
```
static bool convert_symbol(ir_builder_t *ir, ast_t *ast)
{
    if(ast_is_rest(ast) && is_top_level(ir))
    {
        ir_error(ir, "invalid reference to ‘...‘.");
        return false;
    }
    build_ref(ir, ast->symbol);
    return true;
}
```

`convert_node` begins by extracting the first element `fst` from `arr`, which is used to determine the interpretation of those that follow.

⟨*convert_node*⟩≡
```
static bool convert_node(ir_builder_t *ir, ast_t *ast)
{
    ast_array_t *arr = ast->children;
    ast_t *fst = aptr(arr, 0);

    ⟨check token⟩
    switch(fst->token)
    {
        ⟨convert form⟩
    }
    ⟨convert operator⟩
}
```

We treat the `NODE` somewhat like a LISP list, and it's depicted as such. It falls into one of the following cases, depending on the syntactic construct which gave rise to it.

(*expression* [*expressions . . .*])

If not headed by a `TOKEN`, the `ast` represents a *function call*, with the other elements expressions to be evaluated as actual arguments.

⟨*check token*⟩≡
```
if(fst->type != AST_TOKEN)
    return convert_call(ir, fst, arr);
```

Certain `TOKEN`s denote *special forms*. These can introduce control flow, or modify the environment. Their conversion functions may interpret the other elements in `arr` specially – perhaps as lists of declarations, or type specifiers. Converting each is a matter of passing the appropriate elements to its conversion function.

(**let** *declarations*), (**var** *declarations*)

⟨*convert form*⟩≡
```
case LET:
case VAR: assert(alen(arr) == 2);
    return convert_let(ir, aptr(arr, 1), fst->token == LET);
```

(global *declarations***), (const** *declarations***)**

⟨*convert form*⟩+≡
```
case GLOBAL:
case CONST: assert(alen(arr) == 2);
    return convert_global(ir, aptr(arr, 1), fst->token == CONST);
```

(function [*returntype*] **(**[*arguments* ...]**)** *body***)**

⟨*convert form*⟩+≡
```
case FUNCTION: assert(alen(arr) == 4);
    return convert_lambda(ir, rtype_from_spec(aptr(arr, 1)),
                          aptr(arr, 2)->children, aptr(arr, 3));
```

({ [*statements* ...]**)**

⟨*convert form*⟩+≡
```
case '{':
    return convert_prog(ir, arr);
```

(if *condition consequent* [*alternative*]**)**

"`if`" has an optional "`else`".

⟨*convert form*⟩+≡
```
case IF: assert(alen(arr) == 3 || alen(arr) == 4);
    return convert_ifelse(ir, aptr(arr, 1), aptr(arr, 2),
                          (alen(arr) == 4) ? aptr(arr, 3) : NULL);
```

(or *expression expression***), (and** *expression expression***)**

⟨*convert form*⟩+≡
```
case OR: assert(alen(arr) == 3);
    return convert_or(ir, aptr(arr, 1), aptr(arr, 2));
case AND: assert(alen(arr) == 3);
    return convert_and(ir, aptr(arr, 1), aptr(arr, 2));
```

(= *place expression***)**

⟨*convert form*⟩+≡
```
case '=': assert(alen(arr) == 3);
    return convert_set(ir, aptr(arr, 1), aptr(arr, 2));
```

(return *expression***)**

⟨*convert form*⟩+≡
```
case RET: assert(alen(arr) == 2);
    return convert_return(ir, aptr(arr, 1));
```

(while *condition body***)**

A "`while`" loop is just a "`for`" loop without declarations or an iteration expression.

⟨*convert form*⟩+≡
```
case WHILE: assert(alen(arr) == 3);
    return convert_loop(ir, &ast_null, aptr(arr, 1), &ast_null,
                        aptr(arr, 2));
```

(**for** [*declarations*] *condition* [*iteration*] *body*)

⟨*convert form*⟩+≡
```
case FOR: assert(alen(arr) == 5);
    return convert_loop(ir, aptr(arr, 1), aptr(arr, 2), aptr(arr, 3),
                        aptr(arr, 4));
```

(**type** *specifier*)

⟨*convert form*⟩+≡
```
case TYPE: assert(alen(arr) == 2);
    return convert_literal_type(ir, aptr(arr, 1));
```

(**include** *string*)

⟨*convert form*⟩+≡
```
case INCLUDE: assert(alen(arr) == 2);
    return convert_include(ir, aptr(arr, 1));
```

(+ *expression*), (− *expression*)

Unary "`+`" is a no-op. Unary "`-`" becomes a call to "`0-`".

⟨*convert form*⟩+≡
```
case '+':
    if(alen(arr) == 2)
        return convert_expr(ir, aptr(arr, 1));
    /* fallthrough */
case '-':
    if(alen(arr) == 2)
        return convert_token(ir, "0-", arr);
    /* fallthrough */
default:
    break;
```

(*operator* [*expressions* ...])

⟨*convert operator*⟩≡
```
return convert_token(ir, ast_str(fst), arr);
```

If not a special form, the `ast` represents a function call, with `fst` referring to a variable of the same `name` (the synthetic `ast` will be converted to a `REF`.)

⟨*convert_token*⟩≡
```
static inline bool
convert_token(ir_builder_t *ir, const char *name, ast_array_t *arr)
{
    ast_t ast = {
        .type = AST_SYMBOL,
        .symbol = r_intern(name),
    };
    return convert_call(ir, &ast, arr);
}
```

6.5 Bindings

Some special forms bind variables in the global or lexical environments. The ASTs they take as input have a common shape; helper functions exist to simplify their conversion.

⟨*bindings*⟩≡
  ⟨*varbind_t*⟩
  ⟨*bind_from_decl*⟩
  ⟨*binds_from_decls*⟩
  ⟨*binds_finalize*⟩
  ⟨*convert_initexpr*⟩

An array of `varbind_t` structures is populated with the declarations or arguments from an input `NODE`. The `.var` field points to the variable being bound; `.expr` to the initialising expression, if any. Some forms will convert this in a separate pass, setting `.node` to the resulting initial value.

⟨*varbind_t*⟩≡
```
typedef struct
{
    cvar_t *var;
    union
    {
        ast_t *expr;
        cnode_t *node;
    };
} varbind_t;
typedef ARRAY(varbind_t) varbind_array_t;
typedef cvar_t *(*varbind_fn)(rsymbol_t *, rtype_t *, void *);
```

`binds_from_decls` destructures the ASTs from `decls` into `binds`.

⟨*binds_from_decls*⟩≡
```
static void binds_from_decls(ast_array_t *decls, varbind_array_t *binds,
                             varbind_fn varfn, void *data)
{
    int i;

    array_init(binds, alen(decls));
    array_resize(binds, alen(decls));
    array_foreach(decls, i)
        bind_from_decl(aptr(decls, i), aptr(binds, i), varfn, data);
}
```

`bind_from_decl` initialises a `bind` from the `ast` of a declaration.

⟨*bind_from_decl*⟩≡
```
static void bind_from_decl(ast_t *ast, varbind_t *bind, varbind_fn varfn,
                           void *data)
{
    rsymbol_t *name;
    ast_t *tspec = NULL, *expr = NULL;

    if(ast->type == AST_NAME)
    {
        ⟨simple⟩
    }
    else
    {
        ⟨complex⟩
    }
```

```
    ⟨bind⟩
}
```

An input `NAME` is simple – it only specifies the variable `name`.

⟨*simple*⟩≡

```
name = ast->symbol;
```

An input `NODE` has one of the forms (*type name*), (*name init*), or (*type name init*). They're distinguished by the `.type` of the second element `snd`, and the number of `.children`.

⟨*complex*⟩≡

```
assert(ast->type == AST_NODE);
ast_array_t *arr = ast->children;
assert(alen(arr) == 2 || alen(arr) == 3);
ast_t *fst = aptr(arr, 0);
ast_t *snd = aptr(arr, 1);

if(snd->type == AST_NAME)
{
    tspec = fst;
    name = snd->symbol;
    if(alen(arr) == 3)
        expr = aptr(arr, 2);
}
else
{
    name = fst->symbol;
    expr = snd;
}
```

The `.var` field is initialised by the `varfn` callback; taking `name`, type from specifier `tspec`, and supplied `data`. The initialising expression is stored in `.expr`.

⟨*bind*⟩≡

```
*bind = (varbind_t) {
    .var = varfn(name, rtype_from_spec(tspec), data),
    .expr = expr
};
```

The special form conversion function will call `binds_finalize` when it's finished with the `binds` array. If `open` is `true`, conversion is to continue, so any local variables are are added to their functions' lists. Otherwise we are aborting, so they're freed instead.

⟨*binds_finalize*⟩≡

```
static void binds_finalize(varbind_array_t *binds, bool open)
{
    varbind_t *bind;

    array_foreach_ptr(binds, bind)
    {
        if(!cvar_is_global(bind->var))
        {
            cfunction_t *fn = bind->var->local.binder;
            if(open)
                list_add_before(&fn->cvar_head, &bind->var->cvar_list);
            else
                cvar_free(bind->var);
        }
    }
```

```
    array_fini(binds);
}
```

In the absence of an initialising expression, a variable is given an appropriate zero value for its declared type. `convert_initexpr` implements this convention.

⟨*convert_initexpr*⟩≡

```
static bool convert_initexpr(ir_builder_t *ir, varbind_t *bind)
{
    if(bind->expr)
        return convert_expr(ir, bind->expr);
    return build_constant(ir, nil_init(bind->var->decl));
}
```

6.6 Forms

⟨*forms*⟩≡
⟨*let and var*⟩
⟨*globals*⟩
⟨*lambda*⟩
⟨*convert_prog*⟩
⟨*convert_actuals*⟩
⟨*convert_call*⟩
⟨*convert_branch*⟩
⟨*extend_join*⟩
⟨*convert_ifelse*⟩
⟨*convert_or*⟩
⟨*convert_and*⟩
⟨*set_expandp*⟩
⟨*convert_set_expand*⟩
⟨*convert_set*⟩
⟨*convert_return*⟩
⟨*convert_loop*⟩
⟨*convert_loop_exit*⟩
⟨*convert_literal_type*⟩
⟨*convert_file*⟩
⟨*convert_include*⟩

6.6.1 File

The `ast` returned by the parser is a `NODE`, all of whose elements are statements (as opposed to the block passed to `convert_prog`, where the first element is "`{`".)

⟨*convert_file*⟩≡

```
static bool convert_file(ir_builder_t *ir, ast_t *ast)
{
    bool open = true;
    assert(ast->type == AST_NODE);
    for(int i = 0; i < alen(ast->children) && open; i++)
        open = convert_stmt(ir, aptr(ast->children, i));
    return open;
}
```

6.6.2 Let and Var

"`let`" and "`var`" are similar enough that they are converted by a single function.

⟨*let and var*⟩≡
⟨*convert_let_binds*⟩
⟨*convert_var_binds*⟩
⟨*local_create*⟩
⟨*convert_let*⟩

`decls` are first destructured into `binds`, with new variables created by the `local_create` callback. The bindings are then converted by either `convert_let_binds` or `convert_var_binds`, responsible for the semantic difference between the forms. The previous `.value` is restored to `ir`; declarations do not disrupt value flow.

⟨*convert_let*⟩≡
```
static bool convert_let(ir_builder_t *ir, ast_t *decls, bool is_let)
{
    cnode_t *val = ir->value;
    varbind_array_t binds;
    bool open;

    binds_from_decls(decls->children, &binds, local_create, ir);
    open = is_let ?
        convert_let_binds(ir, &binds) :
        convert_var_binds(ir, &binds);
    binds_finalize(&binds, open);
    ir->value = val;
    return open;
}
```

Each variable being introduced is `LEXICAL`, and belongs to the function under conversion. It's optimistically assumed to be a constant – `.is_const` will be reset if an assignment is subsequently seen.

⟨*local_create*⟩≡
```
static cvar_t *local_create(rsymbol_t *name, rtype_t *decl, void *data)
{
    ir_builder_t *ir = data;
    cvar_t *var = cvar_create(name, LEXICAL, decl);

    var->is_const = true;
    var->local.binder = ir->fun->function;
    return var;
}
```

"`let`" assigns initial values to its variables in parallel. The first pass over `binds` converts each initialising expression, saving a pointer to the node that results. This is then associated with its variable by a `BIND` node built in the second pass. The lexical environment is augmented in the second pass so that names in the initial expressions refer to variables outside the scope of the "`let`".

⟨*convert_let_binds*⟩≡

```
static bool convert_let_binds(ir_builder_t *ir, varbind_array_t *binds)
{
    varbind_t *bind;

    array_foreach_ptr(binds, bind)
    {
        if(!convert_initexpr(ir, bind))
            return false;
        bind->node = ir->value;
    }
    array_foreach_ptr(binds, bind)
    {
        build_bind(ir, bind->var, bind->var->decl, bind->node);
        env_add(ir->env, bind->var);
    }
    return true;
}
```

"`var`" assigns initial values to its variables sequentially. A single pass suffices to convert each initialising expression, build the `BIND` node, and augment the environment.

⟨*convert_var_binds*⟩≡

```
static bool convert_var_binds(ir_builder_t *ir, varbind_array_t *binds)
{
    varbind_t *bind;

    array_foreach_ptr(binds, bind)
    {
        if(!convert_initexpr(ir, bind))
            return false;
        build_bind(ir, bind->var, bind->var->decl, ir->value);
        env_add(ir->env, bind->var);
    }
    return true;
}
```

6.6.3 Globals

"`global`" and "`const`" are also converted by a single function.

⟨*globals*⟩≡
⟨*global_create*⟩
⟨*convert_global_binds*⟩
⟨*convert_global*⟩

These forms may only occur at top level. Similar to `convert_let`, declarations are destructured and variables created before IR nodes are built.

⟨*convert_global*⟩≡

```
static bool convert_global(ir_builder_t *ir, ast_t *decls, bool is_const)
{
    cnode_t *val = ir->value;
    varbind_array_t binds;
    bool open;

    if(!is_top_level(ir))
    {
        ir_error(ir, "‘%s‘ only allowed at top level.",
                 is_const ? "const" : "global");
        return false;
    }

    binds_from_decls(decls->children, &binds, global_create, ir);
    open = convert_global_binds(ir, &binds, is_const);
    binds_finalize(&binds, open);
    ir->value = val;
    return open;
}
```

The `global_create` callback first searches the global environment for a variable of given `name`; if not found, it's created and added. It is an error for `var` to have been declared previously, but it may have been referred to by code already converted (and so be present as a `GLOBAL_EXT`.) Regardless, the variable is a `GLOBAL_INT`, with declared type `decl`.

⟨*global_create*⟩≡

```
static cvar_t *global_create(rsymbol_t *name, rtype_t *decl, void *data)
{
    ir_builder_t *ir = data;
    cvar_t *var = env_find(ir->env->global, name);

    if(!var)
        var = env_add_global(ir->env, name);
    else if(var->type == GLOBAL_INT)
        ir_error(ir, "duplicate global declaration ‘%s‘",
                 r_symstr(var->name));
    var->type = GLOBAL_INT;
    var->decl = decl;
    return var;
}
```

Converting the `binds` is a matter of converting each initialising expression (its presence is mandatory when the form is "`const`") and building the `SET` which, when executed, will assign the initial value to the variable. Its `.intl.set` refers to this node; the compiler can make use of it during optimisation; `.is_const` depends on which form is being converted.

⟨*convert_global_binds*⟩≡
```
static bool convert_global_binds(ir_builder_t *ir, varbind_array_t *binds,
                                 bool is_const)
{
    varbind_t *bind;

    array_foreach_ptr(binds, bind)
    {
        if(is_const && !bind->expr)
        {
            ir_error(ir, "`const` must have an initial value.");
            return false;
        }
        if(!convert_initexpr(ir, bind))
            return false;
        bind->var->is_const = is_const;
        bind->var->intl.set = build_set(ir, bind->var, ir->value);
    }
    return true;
}
```

6.6.4 Lambda

A "`function`" expression is converted to a `LAMBDA` node and associated `cfunction_t`.

⟨*lambda*⟩≡
  ⟨*convert_default*⟩
  ⟨*convert_arg_binds*⟩
  ⟨*arg_create*⟩
  ⟨*convert_formals*⟩
  ⟨*convert_lambda*⟩

An `inner` builder is opened, with function and lexical environments nested within the context of `ir`. After checking against maximum arity, the argument bindings are converted; their scope is the body, which is converted next. A `RETURN` is appended if the latter didn't end in one. The `LAMBDA` node is built in the outer context, initialised with the new `cfunction_t` returned by `fun_close`.

⟨*convert_lambda*⟩≡
```
static bool convert_lambda(ir_builder_t *ir, rtype_t *ret_type,
                           ast_array_t *args, ast_t *body)
{
    funenv_t fun;
    lexenv_t env;
    ir_builder_t inner = {
        .fun = fun_open(&fun, ir->fun),
        .env = env_open(&env, ir->env),
        .pfailed = ir->pfailed,
        .file = ir->file
    };

    ir_open(&inner, NULL);
    if(alen(args) > sizeof(argbits_t)*8)
```

```
        ir_error(ir, "too many arguments.");
    else if(convert_formals(&inner, args) && convert_stmt(&inner, body))
        build_return(&inner);
    env_close(&env);
    build_lambda(ir, fun_close(&fun, inner.head, ret_type));
    assert(list_isempty(&fun.cblock_head) || *ir->pfailed);
    return true;
}
```

The first `BIND` node built in the function has a `NULL` variable, and will receive the value of the implicit `argbits` argument containing the argument presence bitflags, as per the calling convention (Chapter 22).

`decls` is destructured to `binds`, as usual. `convert_arg_binds` will build conditionals for evaluating any default expressions, so is passed the `bits` node.

⟨*convert_formals*⟩≡

```
static bool convert_formals(ir_builder_t *ir, ast_array_t *decls)
{
    varbind_array_t binds;
    bool open = true;
    cnode_t *bits = build_bind(ir, NULL, r_type_int, NULL);

    binds_from_decls(decls, &binds, arg_create, ir);
    open = convert_arg_binds(ir, &binds, bits);
    binds_finalize(&binds, open);
    return open;
}
```

The `arg_create` callback creates `LEXICAL` variables (assumed constant, like `local_create`) with the `.is_arg` flag set.

⟨*arg_create*⟩≡

```
static cvar_t *arg_create(rsymbol_t *name, rtype_t *decl, void *data)
{
    ir_builder_t *ir = data;
    cvar_t *var = cvar_create(name, LEXICAL, decl);

    var->is_const = true;
    var->local.binder = ir->fun->function;
    var->local.is_arg = true;
    var->local.is_optional = false;
    return var;
}
```

`convert_arg_binds` proceeds in two passes, similar to `convert_let_binds`. The first pass builds `BIND` nodes in the function entry block, in the same order as the arguments.

If the rest vector "`...`" is present, it must be the last argument, and without declared type or default expression (it's always `.is_optional` with type `vector(object)`.)

When a binding has an initialising expression, this provides the default value for the argument if the latter is omitted in a call; so the variable `.is_optional`.

The second pass builds conditionals for default expressions where needed, and adds the variables to the lexical environment. Later default expressions may reference the values of the arguments which precede them.

⟨*convert_arg_binds*⟩≡

```
static bool convert_arg_binds(ir_builder_t *ir, varbind_array_t *binds,
                              cnode_t *bits)
{
    int i;
```

```
    array_foreach(binds, i)
    {
        varbind_t *bind = aptr(binds, i);

        if(bind->var->name == r_sym_rest)
        {
            if(i != alen(binds)-1 || bind->var->decl || bind->expr)
            {
                ir_error(ir, "invalid '...' declaration.");
                return false;
            }
            bind->var->decl = r_type_vec_object;
            bind->var->local.is_optional = true;
        }
        else if(bind->expr)
        {
            bind->var->local.is_optional = true;
        }
        build_bind(ir, bind->var, bind->var->decl, NULL);
    }
    array_foreach(binds, i)
    {
        varbind_t *bind = aptr(binds, i);

        if(bind->expr && !convert_default(ir, bind, i, bits))
            return false;
        env_add(ir->env, bind->var);
    }
    return true;
}
```

`convert_default` builds a conditional which assigns `bind.expr` to `bind.var` in case the corresponding argument is omitted:

(if (`missing` i $bits$) (= arg_i $expr$)**)**

The `missing BUILTIN` returns `true` when the `i`th bit of `bits` is unset. `i` is constant so, in a somewhat dubious hack, it's boxed, interned, and stashed in the node's `.builtin.optype` field.

Control flow after the conditional is built along the lines of `convert_if`, except that since it's evaluated for effect, there's no need for unifying values with a `PHI` in the join block.

⟨*convert_default*⟩≡

```
extern const cbuiltin_t missing;
static bool convert_default(ir_builder_t *ir, varbind_t *bind, argbits_t i,
                            cnode_t *bits)
{
    ir_builder_t tb, jn;
    robject_t *idx = c_intern(r_box(r_type_int, &i));
    cnode_array_t args;

    array_init(&args, 1);
    array_push(&args, bits);
    build_builtin(ir, &missing, &args, (rtype_t *)idx, r_type_boolean);
    build_if(ir, ir->value);

    ir_open(&tb, ir);
    link_blocks(tb.head, ir->tail);
```

```
    if(!convert_expr(&tb, bind->expr))
        return false;
    build_set(&tb, bind->var, tb.value);

    ir_open(&jn, ir);
    link_blocks(jn.head, tb.tail);
    link_blocks(jn.head, ir->tail);
    ir_extend(ir, &jn);
    return true;
}
```

6.6.5 Block

A nested lexical environment is entered for the duration of a brace-delimited block – the scope of variables introduced by "`let`" and "`var`" is the remainder of the block in which they occur.

Conversion of subexpressions proceeds in order with `convert_stmt`; they're not required to produce values, as none but the last can be used by other nodes. If control flow leaves the block, the loop ends early.

⟨*convert_prog*⟩≡
```
static bool convert_prog(ir_builder_t *ir, ast_array_t *arr)
{
    lexenv_t env;
    bool open = true;
    ir_builder_t inner = *ir;

    inner.env = env_open(&env, ir->env);
    for(int i = 1; i < alen(arr) && open; i++)
        open = convert_stmt(&inner, aptr(arr, i));
    env_close(&env);
    ir_extend(ir, &inner);
    return open;
}
```

6.6.6 Call

The value of the `fst` expression is the `target` of the call. Actual arguments, collected into `args` by `convert_actuals`, may be named; when none are, the `names` array is superfluous and can be emptied. The `CALL` node takes possession of the arrays when built.

⟨*convert_call*⟩≡
```
static bool convert_call(ir_builder_t *ir, ast_t *fst, ast_array_t *arr)
{
    cnode_array_t args;
    rsymbol_array_t names;
    cnode_t *target;
    bool has_names = false;

    if(!convert_expr(ir, fst))
        return false;
    target = ir->value;

    if(!convert_actuals(ir, &args, &names, arr, &has_names))
        return false;
```

```
    if(!has_names)
        array_clear(&names);
    array_shrink(&args);
    array_shrink(&names);
    build_call(ir, target, &args, &names);
    return true;
}
```

Each actual argument has an element at the same index in the `args` and `names` arrays. The number of elements in the call expression is an upper bound on their length. It's an error if there are more arguments than an `argbits_t` can represent.

The elements of `arr` are examined in order. Each may be a `NAME`, `INVALID`, or an arbitrary `ast` denoting an expression.

⟨*convert_actuals*⟩≡
```
static bool
convert_actuals(ir_builder_t *ir, cnode_array_t *args, rsymbol_array_t *names,
                ast_array_t *arr, bool *pnamed)
{
    rsymbol_t *name = NULL;

    array_init(args, alen(arr)-1);
    array_init(names, alen(arr)-1);
    for(int i = 1; i < alen(arr); i++)
    {
        ast_t *ast = aptr(arr, i);

        switch(ast->type)
        {
        case AST_NAME:
            ⟨named⟩
            break;
        case AST_INVALID:
            ⟨omitted⟩
            break;
        default:
            ⟨expression⟩
            break;
        }
    }

    if(alen(args) <= sizeof(argbits_t)*8)
        return true;
    ir_error(ir, "too many arguments.");
fail:
    array_fini(args);
    array_fini(names);
    return false;
}
```

When a `NAME` is encountered, the symbol is stored in `name` to be used by the next element, and `pnamed` is set.

⟨*named*⟩≡
```
*pnamed = true;
name = ast->symbol;
```

INVALID denotes an omitted argument, which is represented with a NULL node and name.

⟨*omitted*⟩≡
```
array_push(args, NULL);
array_push(names, NULL);
```

Anything else is assumed to be an expression and converted as such. The occurrence of the rest vector "..." requests that its elements also be considered as arguments to the call; it's explicitly named to signal this to the backend.

After the node and its name (if any) are appended to the arrays, the latter is reset.

⟨*expression*⟩≡
```
if(!convert_expr(ir, ast))
    goto fail;
if(ast_is_rest(ast))
{
    *pnamed = true;
    name = r_sym_rest;
}
array_push(args, ir->value);
array_push(names, name);
name = NULL;
```

6.6.7 If-Else

An IF node is built with the value of the "if" expression's converted conditional cond. At run-time, when this evaluates to true, the VM will transfer control to the first successor of the block which the node ends; when false, the second. The consequent expression texpr is thus converted (built on branch tb) before the alternative fexpr (branch fb.)

If control flow continues from one or both branches, the ir builder is extended.

⟨*convert_ifelse*⟩≡
```
static bool convert_ifelse(ir_builder_t *ir, ast_t *cond,
                           ast_t *texpr, ast_t *fexpr)
{
    ir_builder_t tb, fb;
    bool topen, fopen;

    if(!convert_expr(ir, cond))
        return false;
    build_if(ir, ir->value);

    topen = convert_branch(&tb, ir, texpr);
    fopen = convert_branch(&fb, ir, fexpr);

    if(topen && fopen)
        extend_join(ir, &tb, &fb);
    else if(topen)
        ir_extend(ir, &tb);
    else if(fopen)
        ir_extend(ir, &fb);
    return topen || fopen;
}
```

A new builder is opened for each branch. The `ast` to be converted may be missing – "`else`" is optional. If it yields no value, a `nil` is synthesised to provide one. A control-flow edge is added from the end of the `ir` builder to the start of the branch `br`.

⟨*convert_branch*⟩≡
```
static inline bool convert_branch(ir_builder_t *br, ir_builder_t *ir,
                                  ast_t *ast)
{
    bool open = true;

    ir_open(br, ir);
    if(ast)
        open = convert_stmt(br, ast);
    if(!br->value)
        build_constant(br, NULL);
    link_blocks(br->head, ir->tail);
    return open;
}
```

Control flow from both branches is merged by `extend_join`. A new builder `jn` is opened and a `PHI` built to merge the values from consequent `tb` and alternative `fb`. Control-flow edges are added, in the same order as the `PHI` arguments, from the end of each branch to the start of the join. The `ir` builder can then be extended.

⟨*extend_join*⟩≡
```
static inline void extend_join(ir_builder_t *ir, ir_builder_t *tb,
                               ir_builder_t *fb)
{
    ir_builder_t jn;

    ir_open(&jn, ir);
    build_phi(&jn, tb->value, fb->value);
    link_blocks(jn.head, tb->tail);
    link_blocks(jn.head, fb->tail);
    ir_extend(ir, &jn);
}
```

6.6.8 And & Or

Short-circuiting "`||`" and "`&&`" are syntactic sugar for "`if`":
(if $left$ `true` $right$**)**

⟨*convert_or*⟩≡
```
static bool convert_or(ir_builder_t *ir, ast_t *lexpr, ast_t *rexpr)
{
    ast_t stub = {
        .type = AST_SYMBOL,
        .symbol = r_intern("true")
    };
    return convert_ifelse(ir, lexpr, &stub, rexpr);
}
```

(if *left right* `false`**)**

⟨*convert_and*⟩≡
```
static bool convert_and(ir_builder_t *ir, ast_t *lexpr, ast_t *rexpr)
{
    ast_t stub = {
        .type = AST_SYMBOL,
        .symbol = r_intern("false")
    };
    return convert_ifelse(ir, lexpr, rexpr, &stub);
}
```

(the `REF` nodes which are built for these will be optimised later to `CONST`s)

6.6.9 Assignment

The left-hand-side `lhs` of an assignment expression must be a `SYMBOL` if it doesn't denote a *set expansion*.

The variable `var` which it names is added as a global if it's not found in the enclosing environment(s). Assigning to a `LEXICAL` variable clears its `.is_const` flag. The right-hand-side expression `rhs` is converted, and a `SET` node built.

⟨*convert_set*⟩≡
```
static bool convert_set(ir_builder_t *ir, ast_t *lhs, ast_t *rhs)
{
    if(set_expandp(lhs))
        return convert_set_expand(ir, lhs->children, rhs);
    if(lhs->type != AST_SYMBOL)
    {
        ir_error(ir, "no set-expansion for expression.");
        return false;
    }

    cvar_t *var = env_find_add(ir->env, lhs->symbol);

    if(!cvar_is_global(var))
        var->is_const = false;
    if(!convert_expr(ir, rhs))
        return false;
    build_set(ir, var, ir->value);
    return true;
}
```

`set_expandp` returns `true` in the case of an `ast` that looks like a call to a named function with at least one argument.

⟨*set_expandp*⟩≡
```
static bool set_expandp(ast_t *ast)
{
    if(ast->type == AST_NODE && alen(ast->children) >= 2)
    {
        ast_t *fst = aptr(ast->children, 0);
        return fst->type == AST_SYMBOL || fst->type == AST_TOKEN;
    }
    return false;
}
```

Set-expansion, as in R (and Lisp,) is syntactic sugar which generates a call to a specially named function in place of the assignment:

(= (`name` *target args* ...) *rhs*) → (`name=` *target rhs args* ...)

A copy of `rhs` is inserted into the left-hand-side `arr` as the third element. `rhs` itself is replaced by `INVALID`, to avoid a double free. The expansion is converted as a call to the function `fst` with "`=`" appended to its `name`.

⟨*convert_set_expand*⟩≡
```
static bool convert_set_expand(ir_builder_t *ir, ast_array_t *arr, ast_t *rhs)
{
    ast_t *fst = aptr(arr, 0), new;
    char *name;

    asprintf(&name, "%s=", ast_str(fst));
    new = (ast_t) {
        .type = AST_SYMBOL,
        .symbol = r_intern(name),
    };
    xfree(name);
    array_insert(arr, 2, *rhs);
    *rhs = (ast_t) { .type = AST_INVALID };
    return convert_call(ir, &new, arr);
}
```

6.6.10 Return

"`return`" builds a `RETURN` node with the converted `expr`, and signals to its caller that control flow has ended.

⟨*convert_return*⟩≡
```
static bool convert_return(ir_builder_t *ir, ast_t *expr)
{
    if(convert_expr(ir, expr))
        build_return(ir);
    return false;
}
```

6.6.11 Loops

"`while`" and "`for`" loops are converted by the same function. Several builders are used; some may be absent or empty, depending on the which input subexpressions are present. After the built blocks are linked together, ignoring back edges, control flows in order:

`prehead` → `header` → `inner` → `footer` → `after`.

⟨*convert_loop*⟩≡
```
static bool convert_loop(ir_builder_t *ir, ast_t *decls, ast_t *cond,
                         ast_t *iter, ast_t *body)
{
    ir_builder_t prehead, header, inner, footer, after;
    lexenv_t env;
    cblock_t *next;
    bool open;
    ⟨convert loop⟩
    open = true;
out:
    if(!ast_is_omitted(decls))
```

```
        env_close(&env);
    return open;
}
```

From the `ir` builder, control flow enters the loop through the pre-header `prehead`. If declarations are present, a new lexical environment `env` is opened to limit their scope to the loop body. Initialising expressions are converted, and variables added, by `convert_let`.

⟨*convert loop*⟩≡

```
ir_open(&prehead, ir);
link_blocks(prehead.head, ir->tail);
if(!ast_is_omitted(decls))
{
    prehead.env = env_open(&env, ir->env);
    open = convert_let(&prehead, decls, true);
    if(!open)
        goto out;
}
```

The loop `header` follows, containing the converted conditional `cond` and ending in an `IF` node. `next` initially points to the header start.

⟨*convert loop*⟩+≡

```
ir_open(&header, &prehead);
open = convert_expr(&header, cond);
next = header.head;
link_blocks(header.head, prehead.tail);
if(!open)
    goto out;
build_if(&header, header.value);
```

A post-loop builder `after` provides a target block for the loop environment.

⟨*convert loop*⟩+≡

```
ir_open(&after, ir);
```

If an iterator subexpression is present, the loop `footer` is opened to contain it, converted as a statement; `next` now points to the footer start.

⟨*convert loop*⟩+≡

```
if(!ast_is_omitted(iter))
{
    ir_open(&footer, &prehead);
    open = convert_stmt(&footer, iter);
    next = footer.head;
}
```

Now the loop environment of the `inner` builder may be initialised: "`break`" entails a jump to the beginning of `after`, and "`continue`" to `next` (either header or footer.)

The body subexpression is converted as a statement; if control flow continues through it, an edge is added from the end of `inner` to `next`. If the iterator is absent, this completes the loop with a back edge.

Otherwise, if control flow both enters and leaves the iterator (as flagged by `open`,) the back edge is built from the end of `footer` to the start of `header`.

⟨*convert loop*⟩+≡
```
ir_open(&inner, &prehead);
inner.loop = &(loopenv_t) { .brk = after.head, .cont = next };
if(convert_stmt(&inner, body))
    link_blocks(next, inner.tail); // body tail -> around again
else
    open = false; // control flow does not enter iter
if(!ast_is_omitted(iter) && open)
    link_blocks(header.head, footer.tail);
```

Control-flow edges may now be added for the conditional which ends `header` – the `true` branch enters the `inner` body; the `false` branch skips to `after`. The `ir` builder extends to this last.

⟨*convert loop*⟩+≡
```
link_blocks(inner.head, header.tail);
link_blocks(after.head, header.tail);
ir_extend(ir, &after);
```

6.6.12 Break & Continue

"`break`" and "`continue`" create control-flow edges to the corresponding blocks in the enclosing loop (assuming one is present.)

⟨*convert_loop_exit*⟩≡
```
static bool convert_loop_exit(ir_builder_t *ir, ast_t *ast)
{
    bool is_break = (ast->token == BREAK);

    if(!ir->loop)
        ir_error(ir, "%s outside loop.",
                 is_break ? "break" : "continue");
    else
        link_blocks(is_break ? ir->loop->brk : ir->loop->cont,
                    ir->tail);
    return false;
}
```

6.6.13 Type

A "type" expression becomes a CONST node with the value of the specified type object. rtype_from_spec returns NULL if the specifier is not valid; here, this is an error.

⟨*convert_literal_type*⟩≡
```
static bool convert_literal_type(ir_builder_t *ir, ast_t *ast)
{
    rtype_t *type = rtype_from_spec(ast);

    if(!type)
    {
        ir_error(ir, "unknown type in ‘type‘ expression.");
        return false;
    }
    build_constant(ir, (robject_t *)type);
    return true;
}
```

6.6.14 Include

After opening an inner builder, continuing from ir, the file named by the argument is read and parsed by p_source. If it fails, an error is signalled, but a message is only emitted if the file couldn't be found (since in case of syntax error, the parser has already done so.) The resulting ast is converted by convert_file.

⟨*convert_include*⟩≡
```
static bool convert_include(ir_builder_t *ir, ast_t *arg)
{
    assert(arg->type == AST_STRING);
    char *filename = arg->string;
    ast_t ast;
    ir_builder_t inner;
    bool open;
    int r;

    assert(filename);
    ir_open(&inner, ir);
    link_blocks(inner.head, ir->tail);
    inner.file = r_intern(filename);
    r = p_source(filename, &ast);
    if(r != 0)
    {
        *ir->pfailed = true;
        if(r < 0)
            ir_error(ir, "including %s - %s", filename,
                     strerror(errno));
        return false;
    }
    open = convert_file(&inner, &ast);
    ast_fini(&ast);
    ir_extend(ir, &inner);
    return open;
}
```

Miscellanea

⟨*includes*⟩≡

```
#include "global.h"
#include "ir.h"
#include "ast.h"
#include "grammar.h"
#include <errno.h>
```

⟨*prototypes*⟩≡

```
static bool convert_expr(ir_builder_t *ir, ast_t *ast);
static bool convert_stmt(ir_builder_t *ir, ast_t *ast);
```

Chapter 7

IR Preparation

This subsystem prepares the IR for later passes by enriching it with derived metadata, enforcing invariants, and removing redundant objects.

⟨*ir_pre.c*⟩≡
  ⟨*includes*⟩
  ⟨*critical edges*⟩
  ⟨*cleanups*⟩
  ⟨*ordering and numbering*⟩
  ⟨*enforce_set*⟩
  ⟨*cfunc_enforce_set*⟩
  ⟨*ir_prepare*⟩

7.1 Prepass

`ir_prepare` is called with the top-level `cfunction_t fn` produced by AST conversion, running its sub-passes recursively over the function and its children.

`cfunc_cleanup` removes redundant blocks, nodes and variables.

`cfunc_crit_edges` ensures that the control-flow graph has no critical edges.

`cfunc_enforce_set` ensures that each value assigned to a variable is of a type consistent with the latter's declaration.

`cfunc_rdfo` numbers and sorts the blocks and nodes in reverse-depth-first order.

`cfunc_init_closure` collects the free variables used in a function and required in its lexical closure; called in postorder.

⟨*ir_prepare*⟩≡
```
cresult ir_prepare(cfunction_t *fn)
{
    cresult res;

    cfunc_cleanup(fn);
    cfunc_crit_edges(fn);
    cfunc_node_users(fn);
    res = cfunc_enforce_set(fn);
    cfunc_rdfo(fn);
    res |= cfunc_map_children(fn, ir_prepare);
    cfunc_init_closure(fn);
    return res;
}
```

7.2 Cleanup

The IR as converted is correct, but may have more objects than strictly necessary – nodes never used, variables not referenced, blocks without nodes. These are detected and removed by `cfunc_cleanup`.

⟨*cleanups*⟩≡
  ⟨*block cleanups*⟩
  ⟨*node and var cleanups*⟩
  ⟨*fix_cleanup*⟩
  ⟨*cfunc_cleanup*⟩

Cleaning up a redundancy can enable the recognition of others, both within and between kinds of object. `cfunc_clean_blocks`, `cfunc_clean_nodes` and `cfunc_clean_vars` are invoked at least once per function `fn`. If any nodes were removed, blocks are examined again.

⟨*cfunc_cleanup*⟩≡
```
void cfunc_cleanup(cfunction_t *fn)
{
    fix_cleanup(fn, cfunc_clean_blocks);
    if(fix_cleanup(fn, cfunc_clean_nodes))
        fix_cleanup(fn, cfunc_clean_blocks);
    cfunc_clean_vars(fn);
}
```

`fix_cleanup` repeatedly calls `clean` until it runs without changing the IR, then returns the accumulated `cresult`.

⟨*fix_cleanup*⟩≡
```
static inline cresult fix_cleanup(cfunction_t *fn,
                                  cresult (*clean)(cfunction_t *))
{
    cresult r, res = SUCCESS;
    do
    {
        r = clean(fn);
        res |= r;
    } while(changed(r));
    return res;
}
```

7.2.1 Blocks

Block cleanups remove empty blocks and merge straight-line sequences, while preserving control-flow connectivity.

⟨*block cleanups*⟩≡
  ⟨*make_room*⟩
  ⟨*splice_preds*⟩
  ⟨*splice_dup_args*⟩
  ⟨*empty_splice*⟩
  ⟨*empty_source*⟩
  ⟨*empty_snip*⟩
  ⟨*coalesce_forward*⟩
  ⟨*update_entry*⟩
  ⟨*select_block_cleanup*⟩
  ⟨*cfunc_clean_blocks*⟩

`cfunc_clean_blocks` examines each `block` in the function; if it's deemed redundant, `select_block_cleanup` returns a a callback `clean` which will make the necessary changes to the IR. Before its invocation, the entry block may need updating; after, the change is noted in the `cresult` flags.

⟨*cfunc_clean_blocks*⟩≡

```
static cresult cfunc_clean_blocks(cfunction_t *fn)
{
    cresult res = SUCCESS;
    cblock_t *block, *btmp;

    list_foreach_entry_safe(&fn->cblock_head, block, btmp, cblock_list)
    {
        blkclean_fn clean = select_block_cleanup(fn, block);
        if(clean)
        {
            update_entry(fn, block);
            clean(block);
            res |= CHANGED;
        }
    }
    return res;
}
```

If the `block` being deleted is the function's `.entry`, the latter pointer must be updated (to its sole successor, which is guaranteed to be present if a cleanup is applicable.)

⟨*update_entry*⟩≡

```
static void update_entry(cfunction_t *fn, cblock_t *block)
{
    if(block == fn->entry)
    {
        assert(alen(&block->succ) == 1);
        fn->entry = aref(&block->succ, 0);
        assert(fn->entry != block);
    }
}
```

A `block` with a single successor `succ` may be eligible for cleanup.

If the block is also empty of nodes, and its removal won't form a critical edge, the applicable cleanup is determined by the number of predecessors it has.

=0 `empty_source`

=1 `empty_snip`

>1 `empty_splice`

If the block isn't empty, and `succ` has only one predecessor (and isn't the entry block,) the former can be coalesced into the latter with `coalesce_forward`.

Self-loops are degenerate but valid, and aren't handled. Empty blocks with no successors are invalid – every control-flow path which terminates does so in a block containing at least a `RETURN` node.

⟨*select_block_cleanup*⟩≡

```
typedef void (*blkclean_fn)(cblock_t *);
static inline blkclean_fn select_block_cleanup(cfunction_t *fn,
                                               cblock_t *block)
{
```

```
    int npred = alen(&block->pred);
    bool empty = list_isempty(&block->cnode_head);

    if(alen(&block->succ) != 1)
        return NULL;

    cblock_t *succ = aref(&block->succ, 0);
    int s_npred = alen(&succ->pred);
    assert(s_npred > 0);

    if(succ == block)
        return NULL;
    if(empty)
    {
        if(npred == 0)
            return empty_source;
        if(npred == 1)
        {
            cblock_t *pred = aref(&block->pred, 0);

            if(alen(&pred->succ) == 1 || s_npred == 1)
                return empty_snip;
        }
        else
        {
            if(alen(&succ->succ) < 2)
                return empty_splice;
        }
    }
    else if(s_npred == 1 && succ != fn->entry)
        return coalesce_forward;
    return NULL;
}
```

`coalesce_forward` coalesces source `block` into its successor `succ`. The predecessor array is copied, backpointers updated in each predecessor and node, then the source block's node list `.cnode_head` is spliced before the destination's.

⟨*coalesce_forward*⟩≡

```
static void coalesce_forward(cblock_t *block)
{
    cblock_t *pred, *succ = aref(&block->succ, 0);
    cnode_t *node;

    array_copy(&succ->pred, &block->pred);
    array_foreach_entry(&succ->pred, pred)
        replace_link(&pred->succ, block, succ);
    list_foreach_entry(&block->cnode_head, node, cnode_list)
        node->block = succ;
    list_splice_after(&succ->cnode_head, &block->cnode_head);
    cblock_free(block);
}
```

`empty_snip` remove the empty `block` between `pred` and `succ`.

⟨*empty_snip*⟩≡
```
static void empty_snip(cblock_t *block)
{
    cblock_t *succ = aref(&block->succ, 0),
             *pred = aref(&block->pred, 0);

    replace_link(&succ->pred, block, pred);
    replace_link(&pred->succ, block, succ);
    cblock_free(block);
}
```

`empty_source` removes the empty block before `succ`; probably the start of the top-level function, since other functions begin with a sequence of `BIND`s.

⟨*empty_source*⟩≡
```
static void empty_source(cblock_t *block)
{
    cblock_t *succ = aref(&block->succ, 0);

    remove_link(&succ->pred, block);
    cblock_free(block);
}
```

The `block` to which `empty_splice` is applied is the `idx`th predecessor of its single successor `succ`. Its predecessors are spliced into the latter's array at this index, and the successor link of each to `block` is redirected.

The `idx`th argument of each `PHI` node in the successor block will enter along all of the newly spliced edges, so it's duplicated to the corresponding elements of the `node`'s argument array.

⟨*empty_splice*⟩≡
```
static void empty_splice(cblock_t *block)
{
    cblock_t *pred, *succ = aref(&block->succ, 0);
    int idx = index_of_block(&succ->pred, block);
    cnode_t *node;

    splice_preds(succ, block, idx);
    array_foreach_entry(&block->pred, pred)
        replace_link(&pred->succ, block, succ);
    list_foreach_entry(&succ->cnode_head, node, cnode_list)
    {
        if(node->type != CN_PHI)
            break;
        splice_dup_args(node, alen(&block->pred), idx);
    }
    cblock_free(block);
}
```

The splice_preds helper makes room for block's predecessor array at index idx in the successor block succ's predecessors, then copies the former into the gap in the latter.

⟨*splice_preds*⟩≡
```
static inline void splice_preds(cblock_t *succ, cblock_t *block, int idx)
{
    cblock_array_t *arr = &succ->pred, *ins = &block->pred;
    int num = alen(ins);

    make_room(arr, num, idx);
    memcpy(arr->ptr + idx, ins->ptr, num * sizeof(cblock_t *));
}
```

The splice_dup_args helper resizes node's .phi.args array, making room for num elements in place of the one at index idx, then duplicates it into each element in the gap, keeping its users updated.

⟨*splice_dup_args*⟩≡
```
static inline void splice_dup_args(cnode_t *node, int num, int idx)
{
    cnode_array_t *arr = &node->phi.args;
    cnode_t *ins = aref(arr, idx);

    make_room(arr, num, idx);
    for(int i = 1; i < num; i++)
    {
        cnode_add_user(node, ins);
        aset(arr, idx + i, ins);
    }
}
```

make_room enlarges the array arr to fit num elements in place of the idxth. The "tail" of elements after idx is moved forward, so it stays at the end of the resized array.

⟨*make_room*⟩≡
```
#define make_room(arr, num, idx) do {                        \
        int len = alen(arr);                                 \
        array_resize(arr, len + num - 1);                    \
        memmove((arr)->ptr + (idx + num),                    \
                (arr)->ptr + (idx + 1),                      \
                (len - idx - 1) * sizeof(*(arr)->ptr));      \
    } while(0);
```

7.2.2 Nodes and Variables

If the value of a node is not used, it may be removed if it has no side effects. Lexical variables which are not mentioned in the program can also be removed.

⟨*node and var cleanups*⟩≡
```
⟨tick⟩
⟨mark_used⟩
⟨mark_var⟩
⟨cfunc_mark_used_nodes⟩
⟨cfunc_clean_unmarked_nodes⟩
⟨cfunc_clean_nodes⟩
⟨cfunc_clean_vars⟩
```

A function will be cleaned up after each significant change to its IR; so a global clock is used, to avoid needing to reset flags.

⟨*tick*⟩≡
```
static unsigned tick = 1;
```

To detect unused objects, each invocation of `cfunc_clean_nodes` increments `tick`. The function `fn` is traversed twice – first to mark objects as used, then to remove those which aren't.

⟨*cfunc_clean_nodes*⟩≡
```
static cresult cfunc_clean_nodes(cfunction_t *fn)
{
    tick++;
    cfunc_mark_used_nodes(fn);
    return cfunc_clean_unmarked_nodes(fn);
}
```

All blocks in the function are assumed (potentially) executable. Each node marks the other nodes and variables that it uses.

⟨*cfunc_mark_used_nodes*⟩≡
```
static void cfunc_mark_used_nodes(cfunction_t *fn)
{
    cblock_t *block;
    cnode_t *node;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        list_foreach_entry(&block->cnode_head, node, cnode_list)
        {
            cnode_map_used(node, mark_used, NULL);
            mark_var(node);
        }
    }
}
```

A node is marked by assigning to its `.mark` field the current value of the global clock. . .

⟨*mark_used*⟩≡
```
static void mark_used(cnode_t **ptr, void *data)
{
    cnode_t *node = *ptr;
    assert(node);
    assert(node->type != CN_SET);
    assert(node->type != CN_IF);
    assert(node->type != CN_RETURN);
    node->mark = tick;
}
```

. . . and a variable, likewise.

⟨*mark_var*⟩≡
```
static void mark_var(cnode_t *node)
{
    switch(node->type)
    {
    case CN_BIND:
    case CN_SET:
        if(node->set.var)
             node->set.var->mark = tick;
        break;
    case CN_REF:
        node->ref.var->mark = tick;
        break;
    default:
        break;
```

```
    }
}
```

An unmarked node will have some value of `.mark` not equal to the clock (assuming no more than `UINT_MAX` increments between cleanup runs.) If it's also pure (Subsection 5.2.4), it's removed and the change recorded.

⟨*cfunc_clean_unmarked_nodes*⟩≡

```
static cresult cfunc_clean_unmarked_nodes(cfunction_t *fn)
{
    cresult res = SUCCESS;
    cblock_t *block;
    cnode_t *node, *ntmp;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        list_foreach_entry_safe(&block->cnode_head, node, ntmp, cnode_list)
        {
            if(node->mark != tick && cnode_is_pure(node, false))
            {
                cnode_remove(node);
                res |= CHANGED;
            }
        }
    }
    return res;
}
```

An unmarked `LEXICAL` variable may be removed if not `.is_closed` (a descendent function requires it, in that case.)

⟨*cfunc_clean_vars*⟩≡

```
static void cfunc_clean_vars(cfunction_t *fn)
{
    cvar_t *var, *tmp;

    list_foreach_entry_safe(&fn->cvar_head, var, tmp, cvar_list)
    {
        if(var->mark != tick && var->type == LEXICAL && !var->local.is_closed)
        {
            assert(!var->local.is_arg);
            cvar_free(var);
        }
    }
}
```

7.3 Critical Edges

An edge in the control-flow graph is *critical* when it joins a block with more than one successor to a block with more than one predecessor. The SSA deconstruction algorithm (Chapter 18) does not correctly place copies in the presence of critical edges, so they're removed by *splitting* each with an empty block (Briggs et al., 1998).

⟨*critical edges*⟩≡
  ⟨*split_edge*⟩
  ⟨*cfunc_crit_edges*⟩

If the edge from `block` to `succ` is critical, it is split and the change noted.

⟨*cfunc_crit_edges*⟩≡

```
cresult cfunc_crit_edges(cfunction_t *fn)
{
    cblock_t *block, *succ;
    cresult res = SUCCESS;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        if(alen(&block->succ) <= 1)
            continue;
        array_foreach_entry(&block->succ, succ)
        {
            if(alen(&succ->pred) <= 1)
                continue;
            split_edge(fn, block, succ);
            res |= CHANGED;
        }
    }
    return res;
}
```

An `empty` block is added to the function, its successor and predecessor arrays initialised, and the links between `block` and `succ` retargeted to splice it into control flow.

⟨*split_edge*⟩≡

```
static void split_edge(cfunction_t *fn, cblock_t *block, cblock_t *succ)
{
    cblock_t *empty = cblock_create();

    list_add(&fn->cblock_head, &empty->cblock_list);
    array_push(&empty->succ, succ);
    array_push(&empty->pred, block);
    replace_link(&block->succ, succ, empty);
    replace_link(&succ->pred, block, empty);
}
```

7.4 Assignment

Conversion to static single assignment form will elide assignments to most lexical variables. Before this occurs, the user's type declarations are given force: at each assignment, the value assigned is constrained to be of the variable's declared type.

⟨*cfunc_enforce_set*⟩≡

```
static cresult cfunc_enforce_set(cfunction_t *fn)
{
    cblock_t *block;
    cnode_t *node;
    cresult res = SUCCESS;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
        list_foreach_entry(&block->cnode_head, node, cnode_list)
            if(node->type == CN_SET || node->type == CN_BIND)
                res |= enforce_set(node, node->set.var);
    return res;
}
```

`enforce_set` also ensures that `.is_const` variables aren't assigned to (except by the initialising expression of a `GLOBAL_INT`.)

⟨*enforce_set*⟩≡
```
static cresult enforce_set(cnode_t *node, cvar_t *var)
{
    if(node->type == CN_SET && var->is_const
       && ((var->type == GLOBAL_INT && var->intl.set != node)
           || var->type == GLOBAL_EXT))
    {
        c_error("invalid assignment to ‘const %s‘", r_symstr(var->name));
        return FAILED;
    }
    if(node->set.value && var->decl)
        return enforce_decl(node, var->decl, &node->set.value, true);
    return SUCCESS;
}
```

7.5 Ordering and Numbering

Some passes must visit the blocks in a function in specific order. To simplify their implementation, the list of blocks is kept sorted in this order. Others store temporary data about IR objects; they can use dense zero-based `.id` numbers to index into out-of-line arrays.

⟨*ordering and numbering*⟩≡
⟨*cblock_number*⟩
⟨*cfunc_number*⟩
⟨*cblock_rdfo*⟩
⟨*cfunc_rdfo*⟩

These invariants are maintained by a call to `cfunc_rdfo` after each pass that adds, removes or rearranges blocks, nodes or functions. It sorts the block list and numbers the objects contained in function `fn`.

⟨*cfunc_rdfo*⟩≡
```
void cfunc_rdfo(cfunction_t *fn)
{
    tick++;
    cblock_rdfo(fn->entry, fn);
    cfunc_number(fn);
}
```

`cblock_rdfo` marks the `block`, visits its unmarked successors, and links it at the function's `.cblock_head` in postorder – on return from the outermost call, the list of (reachable) blocks is sorted in reverse-depth-first order.

⟨*cblock_rdfo*⟩≡
```
static void cblock_rdfo(cblock_t *block, cfunction_t *fn)
{
    cblock_t *succ;

    block->mark = tick;
    array_foreach_entry(&block->succ, succ)
    {
        if(succ->mark != tick)
            cblock_rdfo(succ, fn);
    }
    list_remove(&block->cblock_list);
    list_add(&fn->cblock_head, &block->cblock_list);
}
```

`cfunc_number` counts and numbers the blocks, nodes, and child functions in `fn`. Block and node `.ids` are assigned in order; the `.cfunc_head` list is in no particular order.

⟨*cfunc_number*⟩≡
```
static void cfunc_number(cfunction_t *fn)
{
    cblock_t *block;
    cfunction_t *child;
    int nblocks = 0, nnodes = 0, nfuncs = 0;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        nnodes = cblock_number(block, nnodes);
        block->id = nblocks++;
    }
    fn->nblocks = nblocks;
    fn->nnodes = nnodes;
    assert(nblocks > 0);
    list_foreach_entry(&fn->cfunc_head, child, cfunc_list)
        child->id = nfuncs++;
    fn->nfuncs = nfuncs;
}
```

`cblock_number` counts and numbers the nodes in a `block`, also assigning the node `.ids` at which the latter starts and ends. If the block is empty, it has both these fields set to `INT_MIN`.

⟨*cblock_number*⟩≡
```
static int cblock_number(cblock_t *block, int nnodes)
{
    cnode_t *node;

    if(!list_isempty(&block->cnode_head))
    {
        block->start = nnodes;
        list_foreach_entry(&block->cnode_head, node, cnode_list)
            node->id = nnodes++;
        block->end = nnodes - 1;
    }
    else
        block->start = block->end = INT_MIN;
```

```
    return nnodes;
}
```

7.6 Closures

The lexical variables free in a function are captured by its *closure*, so that they can be accessed even outside the dynamic extent of the function that bound them.

⟨*ir_closure.c*⟩≡
  ⟨*closure includes*⟩
  ⟨*cfunc_populate_closure*⟩
  ⟨*capturep*⟩
  ⟨*add_closed_vars*⟩
  ⟨*add_free_vars*⟩
  ⟨*cfunc_init_closure*⟩

`cfunc_init_closure` determines the variables that a function's lexical closure must capture, and is invoked in a postorder traversal – when called on function `fn`, it has already run on the child functions in its `.cfunc_head` list.

Variables which are required by a child but free in the parent are added to its closure by `add_closed_vars`, and those required by nodes in the function body by `add_free_vars`.

`cfunc_populate_closure` arranges for the values of these captured variables to be copied when the closure is created at run-time (unless `fn` is at top level).

⟨*cfunc_init_closure*⟩≡
```
void cfunc_init_closure(cfunction_t *fn)
{
    add_closed_vars(fn);
    add_free_vars(fn);
    if(fn->node)
        cfunc_populate_closure(fn);
}
```

Each variable `var` required by a `child` is added to the function's `.closure` array if its value is not available (to the child's `LAMBDA`, in this case.)

⟨*add_closed_vars*⟩≡
```
static void add_closed_vars(cfunction_t *fn)
{
    cfunction_t *child;
    cvar_t *var;

    list_foreach_entry(&fn->cfunc_head, child, cfunc_list)
        array_foreach_entry(&child->closure, var)
            if(capturep(var, fn))
                array_push(&fn->closure, var);
}
```

Likewise, every lexical variable `var` to which a `REF` or `SET` node makes reference is added to the `.closure` (and flagged `.is_closed`) if its value isn't available.

⟨*add_free_vars*⟩≡

```
static void add_free_vars(cfunction_t *fn)
{
    cblock_t *block;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        cnode_t *node;

        list_foreach_entry(&block->cnode_head, node, cnode_list)
        {
            cvar_t *var;

            if(node->type == CN_REF)
                var = node->ref.var;
            else if(node->type == CN_SET)
                var = node->set.var;
            else
                continue;
            if(cvar_is_global(var) || !capturep(var, fn))
                continue;
            var->local.is_closed = true;
            array_push(&fn->closure, var);
        }
    }
}
```

A lexical variable `var` has its value available in the body of a function if it's bound by the function, or if it's free but already in the function's `.closure`. `capturep` returns `false` in such cases.

⟨*capturep*⟩≡

```
static inline bool capturep(cvar_t *var, cfunction_t *fn)
{
    cvar_t *chk;

    if(var->local.binder == fn)
        return false;
    array_foreach_entry(&fn->closure, chk)
        if(chk == var)
            return false;
    return true;
}
```

When a `LAMBDA` node is executed it creates an `rclosure_t` (Section 21.4) into which it copies the values of the nodes in its `.lambda.closure` array. These are used, when the closure is `CALL`ed, to satisfy accesses to the corresponding variables in the function's `.closure`.

Each `.is_const` variable in the latter has a `REF` node inserted just before the `LAMBDA`, if it doesn't have one already. This is stored at the corresponding index in `.lambda.closure` (and its `.users` kept up-to-date)

Mutable variables – i.e. `!.is_const` – will be dealt with later (Chapter 12), after which another call will be made to `cfunc_populate_closure`, hence the check for `*pnode`.

⟨*cfunc_populate_closure*⟩≡
```
cresult cfunc_populate_closure(cfunction_t *fn)
{
    cresult res = SUCCESS;
    cnode_t *def = fn->node;
    int i;

    array_resize(&def->lambda.closure, alen(&fn->closure));
    array_foreach(&fn->closure, i)
    {
        cvar_t *var = aref(&fn->closure, i);
        cnode_t **pnode = aptr(&def->lambda.closure, i), *node;

        if(!var->is_const || *pnode)
            continue;

        node = cnode_insert_before(def, CN_REF);
        node->ref.var = var;
        cnode_add_user(def, node);
        *pnode = node;
        res |= CHANGED;
    }
    return res;
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
```

⟨*closure includes*⟩≡
```
#include "global.h"
#include "ir.h"
```

Chapter 8

Dominator Tree

Several properties of the control flow graph of a function, and the basic blocks (Subsection 5.1.3) that it contains, are informed by the concept of dominance.

If the block `x` appears on every control-flow path from the function entry to the block `y`, then `x` *dominates* `y`. If, in addition, `x != y`, then `x` *strictly dominates* `y`. `x` is the *immediate dominator* of `y` if `x` strictly dominates `y`, and every other dominator of `y` dominates `x` (Lengauer and Tarjan, 1979; Cytron et al., 1991).

The *dominator tree* of a function is rooted at the `.entry` block, and every other block `y` has its immediate dominator as its parent in the tree (Cytron et al., 1991).

The *dominance frontier* of a block `x` is the set of all blocks `y` such that `x` dominates some predecessor of `y` but doesn't strictly dominate `y` (Cytron et al., 1991).

⟨*ir_dom.c*⟩≡
⟨*includes*⟩
⟨*intersect*⟩
⟨*dom_idoms*⟩
⟨*add_df*⟩
⟨*dom_collect*⟩
⟨*dom_free*⟩
⟨*cfunc_dom*⟩

The SSA conversion (Chapter 9) and value numbering (Chapter 14) passes rely on the dominator tree to efficiently determine the definition that reaches a use. The SSA conversion pass uses the dominance frontier to place ϕ-functions where the values from different assignments of a variable merge together.

The `doms_t` structure stores these sets. The element in `.dfs` indexed by a block's `.id` holds an array containing references to the blocks in its dominance frontier. The element at the same index in `.children` holds an array of references to its children in the dominator tree.

⟨*doms_t*⟩≡
```
typedef struct
{
    cblock_array_t *dfs;
    cblock_array_t *children;
} doms_t;
```

`cfunc_dom` populates the immediate dominator array `idom`, then derives from that the dominator tree and dominance frontier sets.

⟨*cfunc_dom*⟩≡
```
doms_t *cfunc_dom(cfunction_t *fn)
{
    doms_t *dom;
    cblock_t **idom = dom_idoms(fn);

    dom = dom_collect(fn, idom);
    xfree(idom);
    return dom;
}
```

The `doms_t` returned can be deallocated with `dom_free`.

⟨*dom_free*⟩≡
```
void dom_free(doms_t *dom, int nblocks)
{
    for(int i=0; i<nblocks; i++)
    {
        array_fini(&dom->children[i]);
        array_fini(&dom->dfs[i]);
    }
    xfree(dom->children);
    xfree(dom->dfs);
    xfree(dom);
}
```

8.1 Immediate Dominators

A straightforward implementation of the algorithm described in Cooper, Harvey, and Kennedy (2011, Figure 3), `dom_idoms` requires that blocks in the input function `fn` have been numbered in postorder (a task performed by `cfunc_rdfo`.)

The immediate dominator of a `block` is its predecessor, if it has just the one; or the nearest block which dominates all of its predecessors. It's stored in the `doms` array, indexed by block `.id`. The `.entry` block is its own immediate dominator, to ground the `intersect` calls. The algorithm iterates to convergence.

⟨*dom_idoms*⟩≡
```
static cblock_t **dom_idoms(cfunction_t *fn)
{
    cblock_t **doms = xcalloc(fn->nblocks, sizeof(*doms));
    bool changed = (fn->nblocks > 1);

    doms[0] = fn->entry;
    while(changed)
    {
        cblock_t *block;

        changed = false;
        list_foreach_entry(&fn->cblock_head, block, cblock_list)
        {
            cblock_t *pred, *new_idom = NULL;

            if(block == fn->entry)
                continue;
            array_foreach_entry(&block->pred, pred)
```

```
            {
                if(doms[pred->id])
                {
                    if(!new_idom)
                        new_idom = pred;
                    else
                        new_idom = intersect(doms, pred, new_idom);
                }
            }
            if(doms[block->id] != new_idom)
            {
                doms[block->id] = new_idom;
                changed = true;
            }
        }
    }
    return doms;
}
```

The sequence `doms[x]`, `doms[doms[x]]`, ... corresponds to the path up the dominator tree from block `x` to the entry. `intersect` advances `b1` and `b2` until a common ancestor is found.

⟨*intersect*⟩≡

```
static cblock_t *intersect(cblock_t *doms[], cblock_t *b1, cblock_t *b2)
{
    while(b1 != b2)
    {
        while(b1->id > b2->id)
            b1 = doms[b1->id];
        while(b2->id > b1->id)
            b2 = doms[b2->id];
    }
    return b1;
}
```

8.2 Dominator Tree & Dominance Frontier

Other passes actually use the inverse of `doms` – for each block in the dominator tree, they require a list of the children, not the parent. These are initialised by `dom_collect`, which additionally computes the dominance frontier sets in the manner described by Cooper et al. (2011, Figure 5).

⟨*dom_collect*⟩≡

```
static doms_t *dom_collect(cfunction_t *fn, cblock_t *doms[])
{
    cblock_array_t *dfs = xcalloc(fn->nblocks, sizeof(*dfs));
    cblock_array_t *children = xcalloc(fn->nblocks, sizeof(*children));
    doms_t *dom = xmalloc(sizeof(*dom));
    cblock_t *block;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        if(block == fn->entry)
            continue;
        array_push(&children[doms[block->id]->id], block);
        if(alen(&block->pred) > 1)
        {
```

```
            cblock_t *pred;

            array_foreach_entry(&block->pred, pred)
            {
                cblock_t *runner = pred;
                while(runner != doms[block->id])
                {
                    add_df(block, &dfs[runner->id]);
                    runner = doms[runner->id];
                }
            }
        }
    }
    *dom = (doms_t) {
        .dfs = dfs,
        .children = children,
    };
    return dom;
}
```

`add_df` will not insert `block` into `set` more than once. It could be improved with e.g. a presence bitvector if quadratic running time is to be avoided; it hasn't been a problem in practice.

⟨*add_df*⟩≡

```
static void add_df(cblock_t *block, cblock_array_t *set)
{
    cblock_t *tmp;

    array_foreach_entry(set, tmp)
        if(tmp == block)
            return;
    array_push(set, block);
}
```

Miscellanea

⟨*includes*⟩≡

```
#include "global.h"
#include "ir.h"
```

⟨*ir_dom.h*⟩≡

```
⟨doms_t⟩
doms_t *cfunc_dom(cfunction_t *fn);
void dom_free(doms_t *dom, int nblocks);
```

Chapter 9

SSA Form Construction

A program in *static single assignment* form has exactly one assignment to each variable, and that assignment dominates all uses of the variable.

SSA form allows the representation of a program's data flow information in a more compact form, rendering analyses and optimisations simpler and more efficient (Kelsey, 1995; Briggs et al., 1998; Cytron et al., 1991).

⟨*ir_ssa.c*⟩≡
  ⟨*includes*⟩
  ⟨*phi_t*⟩
  ⟨*defsrc_t*⟩
  ⟨*def_t*⟩
  ⟨*globals*⟩
  ⟨*place_phi*⟩
  ⟨*insert_phis*⟩
  ⟨*lookup_def*⟩
  ⟨*has_var*⟩
  ⟨*push_def*⟩
  ⟨*ssa_convert*⟩
  ⟨*finalize_phis*⟩
  ⟨*collect_vars*⟩
  ⟨*cfunc_ssa_convert*⟩
  ⟨*ir_ssa_convert*⟩

Where different control-flow paths join, a pseudo-function called a ϕ-*function* introduces a definition representing the merging of values from multiple assignments (Briggs et al., 1998).

During conversion to SSA form, ϕ-functions may be inserted at the start of blocks reached by more than one assignment to a variable. Their provisional presence is recorded in a list of `phi_t` structures, one for each `.var` being converted, singly linked through the `.next` field. The `PHI .node` itself will only be added to the block if it's `LIVE`.

⟨*phi_t*⟩≡
```
typedef struct phi
{
    cnode_t *node;
    cvar_t *var;
    enum { PENDING, DEAD, LIVE } state;
    struct phi *next;
} phi_t;
```

The source of a definition is either some `NODE` in the input or a provisional `PHI`, recorded by a `defsrc_t` structure.

⟨*defsrc_t*⟩≡
```
typedef struct
{
    enum { PHI, NODE } type;
    union
    {
        phi_t *phi;
        cnode_t *node;
    };
} defsrc_t;
```

As conversion proceeds, and a variable's definitions encountered, `def_t` structures are pushed onto a stack. `.src` records the variable's value at its assignment in `.block`; `.next_stack` points to the stack upon which another definition in the same block was pushed.

⟨*def_t*⟩≡
```
typedef struct def def_t;
typedef def_t *def_stack_t;
typedef struct def
{
    defsrc_t src;
    cblock_t *block;
    def_stack_t *next_stack;
    def_t *next;
} def_t;
```

Each element of the `defs` array is a pointer to the top definition on the stack for the variable with corresponding `.id`. In a similar manner, each element of `phis`, indexed by block `.id`, points to the head of the list of `phi_t` records provisionally inserted in that block. `vars` contains the variables being considered for conversion.

⟨*globals*⟩≡
```
static def_stack_t *defs;
static phi_t **phis;
static cvar_array_t vars;
```

To recursively convert `fn` and its child functions into SSA form, the compiler will invoke `ir_ssa_convert`.

⟨*ir_ssa_convert*⟩≡
```
void ir_ssa_convert(cfunction_t *fn)
{
    cfunc_ssa_convert(fn);
    cfunc_mapc_children(fn, ir_ssa_convert);
}
```

`cfunc_ssa_convert` operates on the single function `fn`, using the dominator tree and dominance frontier sets returned by `cfunc_dom` (Chapter 8). Variables eligible for conversion are recorded and numbered by `collect_vars`. `defs` and `phis` are allocated; one stack per variable, one list per block.

`insert_phis` adds provisional ϕ-functions as `phi_t` records to the latter lists, then `ssa_convert` "renames" variables via the former, with unique `cnode_t`s taking the place of shared `rsymbol_t`s.

The ϕ-functions discovered to be `LIVE` are linked into the IR by `finalize_phis`. The graph has likely been modified, so after deallocating intermediate structures, calls to `cfunc_cleanup` and `cfunc_rdfo` prune redundant objects and number any newly added `PHI`s.

⟨*cfunc_ssa_convert*⟩≡
```
void cfunc_ssa_convert(cfunction_t *fn)
{
    doms_t *dom = cfunc_dom(fn);

    collect_vars(fn, &vars);
    defs = xcalloc(vars.length, sizeof(*defs));
    phis = xcalloc(fn->nblocks, sizeof(*phis));
    insert_phis(dom, fn);
    ssa_convert(dom, fn->entry);
    finalize_phis(fn);
    xfree(phis);
    xfree(defs);
    array_fini(&vars);
    dom_free(dom, fn->nblocks);
    cfunc_cleanup(fn);
    assert(!cfunc_crit_edges(fn));
    cfunc_rdfo(fn);
}
```

Since the values flowing between `cnode_t`s never leave the body of a single `cfunction_t`, only the lexical variables bound by the given `fn` undergo SSA conversion. Another call to `cfunc_ssa_convert` will be made by a later pass to convert variables which require cells, after they've been introduced; they're ignored for now.

⟨*collect_vars*⟩≡
```
static void collect_vars(cfunction_t *fn, cvar_array_t *vars)
{
    int nvars = 0;
    cvar_t *var;

    array_init(vars, 0);
    list_foreach_entry(&fn->cvar_head, var, cvar_list)
    {
        if(cvar_is_global(var) || cvar_is_celled(var))
            continue;
        var->id = nvars++;
        array_push(vars, var);
    }
}
```

For all and only the variables undergoing conversion, `has_vars` returns `true`.

⟨*has_var*⟩≡
```
static inline bool has_var(cvar_t *var)
{
    cvar_t *v;

    array_foreach_entry(&vars, v)
        if(v == var)
            return true;
    return false;
}
```

9.1 ϕ-function Placement

`insert_phis` implements the algorithm given in Cytron et al. (1991, Figure 11). Each variable `var` requires ϕ-functions placed in the iterated dominance frontier of the blocks in which its assignments are found. To avoid clearing the `added` and `processed` flag arrays between variables, a clock `tick` is used, as in Subsection 7.2.2.

⟨*insert_phis*⟩≡
```
static void insert_phis(doms_t *dom, cfunction_t *fn)
{
    ARRAY(cblock_t *) work = ARRAY_INIT;
    unsigned *added = xcalloc(fn->nblocks, sizeof(unsigned));
    unsigned *processed = xcalloc(fn->nblocks, sizeof(unsigned));
    unsigned tick = 0;
    cblock_t *block, *wkblk;
    cnode_t *node;
    cvar_t *var;

    array_foreach_entry(&vars, var)
    {
        tick++;
        ⟨place phis⟩
    }
    xfree(added);
    xfree(processed);
    array_fini(&work);
}
```

The worklist `work` is initialised with each `block` containing at least one `SET` or `BIND` node assigning to `var`. The `processed` flag is set on each such block.

⟨*place phis*⟩≡
```
list_foreach_entry(&fn->cblock_head, block, cblock_list)
{
    unsigned id = block->id;

    list_foreach_entry(&block->cnode_head, node, cnode_list)
    {
        if((node->type == CN_SET || node->type == CN_BIND)
           && node->set.var == var)
        {
            processed[id] = tick;
            array_push(&work, block);
            break;
        }
```

```
    }
}
```

Until the workqueue is empty, a block `wkblk` is taken. For every `block` in its dominance frontier, a `phi` is created and `added` to the `phis` array, if one hasn't been already.

The value resulting from this merge is interpreted as being assigned to the variable, so may itself require the placement of further ϕ-functions – if the block hasn't been `processed` yet, it's added to the worklist.

⟨*place phis*⟩+≡

```
while(array_take(&work, &wkblk))
{
    array_foreach_entry(&dom->dfs[wkblk->id], block)
    {
        unsigned id = block->id;

        if(added[id] < tick)
        {
            place_phi(block, var);
            added[id] = tick;
            if(processed[id] < tick)
            {
                processed[id] = tick;
                array_push(&work, block);
            }
        }
    }
}
```

A `phi_t` record for `var` is placed in a `block` by adding it to the corresponding element of the `phis` array. A `PHI` node is created and stored in the `.node` field, with one argument per control-flow inedge, but it's not added to the block's `.cnode_head` list – some of these ϕ-functions may be unnecessary in the resulting code, but we don't yet know which; its `.state` is `PENDING`.

⟨*place_phi*⟩≡

```
static void place_phi(cblock_t *block, cvar_t *var)
{
    unsigned len = alen(&block->pred);
    phi_t *phi = xmalloc(sizeof(*phi));
    cnode_t *node = cnode_create(block, CN_PHI);

    *phi = (phi_t) {
        .node = node,
        .var = var,
        .state = PENDING
    };
    array_init(&node->phi.args, len);
    array_resize(&node->phi.args, len);
    slist_push(phis[block->id], phi, next);
}
```

9.2 Variable Renaming

`ssa_convert` implements the algorithm of Cytron et al. (1991, Figure 12) and Briggs et al. (1998, Figure 3, step 2). Instead of using an array C of counters to rename the variable at each assignment, the `cnode_t` yielding the value assigned becomes the unique “name”, and is substituted in place of the corresponding `REF`s.

⟨*ssa_convert*⟩≡
```
static void ssa_convert(doms_t *dom, cblock_t *block)
{
    phi_t *phi;
    cnode_t *node, *tmp;
    cblock_t *nextblk;
    def_stack_t *unwind = NULL;

    ⟨convert block⟩
}
```

ϕ-functions come before the other nodes in the block. Each `phi_t` added earlier counts as an assignment, so a `defsrc_t` recording the `phi` is pushed onto its variable’s stack.

⟨*convert block*⟩≡
```
slist_foreach(phis[block->id], phi, next)
{
    defsrc_t src = { .type = PHI, .phi = phi };

    unwind = push_def(phi->var, src, block, unwind);
}
```

The nodes in the block are visited in control-flow order. A `SET` or `BIND` of a variable being converted is an assignment, so a `defsrc_t` for the assigned value is prepared (recall that a `BIND` with `NULL .set.value` introduces a function argument, and its value is taken directly.)

The first optimisation of Briggs et al. (1998, Section 4) applies here: if there’s a definition of `var` earlier in the block, the record on the stack can just be replaced instead of pushing a new one.

A `SET` or local `BIND` can be safely removed at this point as, by construction, no other nodes use their values.

A `REF` node of such a variable is reached by the definition returned by `lookup_def` – which replaces the `node` in its users. Its purpose fulfilled, the `REF` may now be freed.

⟨*convert block*⟩+≡
```
list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
{
    if((node->type == CN_SET || node->type == CN_BIND)
           && has_var(node->set.var))
    {
        cvar_t *var = node->set.var;
        def_t *tos = defs[var->id];
        defsrc_t src = {
            .type = NODE,
            .node = node->set.value ? node->set.value : node
        };

        if(tos && tos->block == node->block)
            tos->src = src;
        else
            unwind = push_def(var, src, block, unwind);
        if(node->set.value)
```

```
            cnode_remove(node);
    }
    else if(node->type == CN_REF && has_var(node->ref.var))
    {
        cnode_t *value = lookup_def(node->ref.var);
        assert(value);
        cnode_replace_in_users(node, value);
        cnode_free(node);
    }
}
```

A different definition reaches a ϕ-function along each control-flow inedge. In each successor `nextblk`, each `phi_t` is given, as its argument at corresponding index `i`, the definition of its variable that reaches the end of the current `block`.

Minimal (unpruned) SSA form inserts more ϕ-functions than necessary – in particular, outside the scope of some variables, such as those declared in a loop body. The control-flow edge bypassing the loop carries no meaningful value, and no references can occur beyond it. Any `phi_t` using such a value is redundant, its `.state` set to `DEAD`.

⟨*convert block*⟩+≡

```
array_foreach_entry(&block->succ, nextblk)
{
    int i = index_of_block(&nextblk->pred, block);

    slist_foreach(phis[nextblk->id], phi, next)
    {
        cnode_t *value = lookup_def(phi->var);

        if(value)
            aset(&phi->node->phi.args, i, value);
        else
            phi->state = DEAD;
    }
}
```

The block's processing completed, we recursively proceed to its children in the dominator tree.

⟨*convert block*⟩+≡

```
array_foreach_entry(&dom->children[block->id], nextblk)
    ssa_convert(dom, nextblk);
```

After those are processed, the definition stacks are unwound, as per the second optimisation of Briggs et al. (1998, Section 4). Popping the stacks on the `unwind` list removes just those definitions that were pushed by this block.

⟨*convert block*⟩+≡

```
while(unwind)
{
    def_t *def = *unwind;

    slist_remove_head(*unwind, next);
    unwind = def->next_stack;
    xfree(def);
}
```

When the value of `src` is assigned to a variable `var` in a `block`, a new definition `def` is allocated, initialised, and pushed onto the corresponding `stack` by `push_def`.

`unwind` heads a linked list of stacks, onto which `block` has pushed definitions. `.next_stack` takes the old value, and the new value is returned.

⟨*push_def*⟩≡
```
static def_stack_t *push_def(cvar_t *var, defsrc_t src,
                             cblock_t *block, def_stack_t *unwind)
{
    def_stack_t *stack = &defs[var->id];
    def_t *def = xmalloc(sizeof(*def));

    *def = (def_t) {
        .src = src,
        .block = block,
        .next_stack = unwind
    };
    slist_push(*stack, def, next);
    return stack;
}
```

`lookup_def` returns the definition at the top of the stack for variable `var`. If it's a provisional `PHI`, it's confirmed to be `LIVE` unless we know otherwise.

⟨*lookup_def*⟩≡
```
static cnode_t *lookup_def(cvar_t *var)
{
    def_t *def = defs[var->id];

    if(!def)
        return NULL;
    if(def->src.type == PHI)
    {
        phi_t *phi = def->src.phi;

        if(phi->state != DEAD)
        {
            phi->state = LIVE;
            return phi->node;
        }
        return NULL;
    }
    return def->src.node;
}
```

9.3 ϕ-function Realization

Conversion complete, a `phi_t`'s record now correctly indicates its `.state`. Those that are `LIVE` have their `.node`s linked into the IR (at the head of their `block`s) and record themselves as using their arguments.

⟨*finalize_phis*⟩≡

```
static void finalize_phis(cfunction_t *fn)
{
    cblock_t *block;
    phi_t *phi, *tmp;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        slist_foreach_safe(phis[block->id], phi, tmp, next)
        {
            if(phi->state == LIVE)
            {
                cnode_t *arg;

                array_foreach_entry(&phi->node->phi.args, arg)
                    cnode_add_user(phi->node, arg);
                list_add(&block->cnode_head, &phi->node->cnode_list);
            }
            else
                cnode_free(phi->node);
            xfree(phi);
        }
    }
}
```

Miscellanea

⟨*includes*⟩≡

```
#include "global.h"
#include "ir.h"
```

Chapter 10

Analysis & Optimisation

Optimising a program is the process of improving its efficiency while retaining its meaning. This subsystem performs data-flow analysis using the *sparse conditional constant propagation* algorithm described in Wegman and Zadeck (1991), which enables type recovery, constant expression evaluation, and dead code elimination (Aho et al., 2006, Section 9.3).

⟨*opt_sccp.c*⟩≡
  ⟨*sccp includes*⟩
  ⟨*context*⟩
  ⟨*sccp*⟩
  ⟨*optimise*⟩
  ⟨*cfunc_closure_pre*⟩
  ⟨*cfunc_closure_post*⟩
  ⟨*ir_optimise*⟩

⟨*opt_trans.c*⟩≡
  ⟨*trans includes*⟩
  ⟨*builtins*⟩
  ⟨*transfer*⟩
  ⟨*transform*⟩

Types and constant values are discovered and propagated through each function in the input. Some builtin operations can be computed ahead of time if the values of their arguments are known. `CALL`s to functions of known type can match named arguments and become `_FAST`; `CALL`s to known `LAMBDA`s can be expanded directly inline. Nodes annotated with recovered types admit the generation of specialised code (which executes faster, without the overhead of unboxing or type-directed dispatch.) Blocks which are not reachable on any execution path are detected and removed from the program.

10.1 Entry Point

The subsystem entry point, `ir_optimise`, repeatedly analyses and optimises the function `fn` until a fixed point is reached, before moving on to its lexical children.

Before entering the loop, `cfunc_closure_pre` updates the types of closed variables with information from the function's `.parent`.

Each iteration first recovers type and constant information with `cfunc_sccp`, then transforms the program accordingly with `cfunc_optimise`, expanding invocations of known functions with `inline_lambdas`.

The result flags for the optimisation pass are set in `ores`; those for just the inlining phase in `ires`. If the IR was `changed`, it's cleaned and renumbered.

After inlining, some copied variables may no longer be captured by a lexical closure; `cfunc_closure_post` finds them and clears their `.is_closed` flags. Another call to `cfunc_ssa_convert` completes the integration of inlined code.

⟨*ir_optimise*⟩≡
```
cresult ir_optimise(cfunction_t *fn)
{
    cresult res = SUCCESS, ores, ires;

    if(fn->parent)
        cfunc_closure_pre(fn);
    do
    {
        if(fn->nnodes == 0 || fn->nblocks == 0)
            break;

        opt_init(fn);
        cfunc_sccp(fn);
        ores = cfunc_optimise(fn);
        res |= ores |= ires = inline_lambdas(fn, &inlines);
        opt_fini();
        if(ores == CHANGED)
        {
            cfunc_cleanup(fn);
            cfunc_rdfo(fn);
        }
        if(ires == CHANGED)
        {
            cfunc_closure_post(fn);
            cfunc_ssa_convert(fn);
        }
    } while(ores == CHANGED);
    return res | cfunc_map_children(fn, ir_optimise);
}
```

Optimisation of closed variables is complicated by their representation – after SSA conversion, other lexical variables are implicit in the value flow, whereas closed variables are still explicitly accessed through `REF` and `SET` nodes. A more uniform treatment would be desirable, perhaps via the "lambda-lifting" transformation described in Clinger and Hansen (1994).

`cfunc_closure_pre` assigns to each variable that `.is_const` in the function `fn`'s `.closure` the type – recovered or declared – of the corresponding value captured by the function's `LAMBDA` node in its lexical parent (Section 7.6).

⟨*cfunc_closure_pre*⟩≡
```
static void cfunc_closure_pre(cfunction_t *fn)
{
    cvar_array_t *vars = &fn->closure;
    cnode_array_t *vals = &fn->node->lambda.closure;
    int i;

    array_foreach(vars, i)
    {
        cvar_t *var = aref(vars, i);
        cnode_t *val = aref(vals, i);

        if(!val || !var->is_const)
            continue;
        var->decl = val->decl;
```

```
    }
}
```

`cfunc_closure_post` clears the `.is_closed` flags of the variables bound by the function `fn`, then sets it for only those actually needed by its lexical `child` functions. Closed variables copied from inlined functions (Chapter 11) can then participate in SSA conversion (Chapter 9) while avoiding unnecessary cell introduction (Chapter 12).

⟨*cfunc_closure_post*⟩≡

```
static void cfunc_closure_post(cfunction_t *fn)
{
    cfunction_t *child;
    cvar_t *var;

    list_foreach_entry(&fn->cvar_head, var, cvar_list)
    {
        if(cvar_is_global(var))
            continue;
        var->local.is_closed = false;
    }
    list_foreach_entry(&fn->cfunc_head, child, cfunc_list)
    {
        array_foreach_entry(&child->closure, var)
            var->local.is_closed = true;
    }
}
```

10.2 Context

⟨*context*⟩≡
⟨*globals*⟩
⟨*opt_init*⟩
⟨*opt_fini*⟩
⟨*cell_meet*⟩
⟨*cell_changed*⟩

Each node (Subsection 5.1.4) in the program has a corresponding LatticeCell in the `cells` array, representing compile-time knowledge about the constant value (and type of object) that it yields during execution (Wegman and Zadeck, 1991, Subsection 2.2).

⟨*globals*⟩≡

```
cell_t *cells;
```

Each block (Subsection 5.1.3) has a corresponding ExecutableFlag in `flags`. When this is `true`, control flow from the function `.entry` can reach the block along an inedge from one of its predecessors.

Since critical edges are not present in the control flow graph (Section 7.3), a single flag can be unambiguously associated with each block (Wegman and Zadeck, 1991, Subsection 5.3).

⟨*globals*⟩+≡

```
bool *flags;
```

These arrays are indexed by node or block `.id` via helper functions.

⟨*global helpers*⟩≡

```
static inline cell_t *cell_for(cnode_t *node)
    { return &cells[node->id]; }
static inline bool flag_for(cblock_t *block)
    { return flags[block->id]; }
```

The SCCP algorithm consumes items from two worklists. `val_work` holds pointers to nodes which need to be (re)analysed; it corresponds to the SSAWorkList of Wegman and Zadeck (1991, Subsection 3.4). `ctrl_work` has pointers to blocks found executable but yet to be visited, and corresponds to FlowWorkList.

⟨*globals*⟩+≡
```
static WORKLIST(cnode_t *) val_work;
static WORKLIST(cblock_t *) ctrl_work;
```

During optimisation, call sites which are suitable candidates for inlining are accumulated in the `inlines` array.

⟨*globals*⟩+≡
```
cnode_array_t inlines;
```

The global arrays are freshly initialised for each iteration of the analysis-optimisation loop.

⟨*opt_init*⟩≡
```
static inline void opt_init(cfunction_t *fn)
{
    cells = xcalloc(fn->nnodes, sizeof(*cells));
    flags = xcalloc(fn->nblocks, sizeof(*flags));
    array_init(&inlines, 0);
}
```

The function may change shape during `cfunc_optimise`, so it's not safe to reuse `cells` or `flags`.

⟨*opt_fini*⟩≡
```
static inline void opt_fini()
{
    array_fini(&inlines);
    xfree(cells);
    xfree(flags);
}
```

10.2.1 LatticeCells

The `cell_t` structure represents a LatticeCell. The `.state` field specifies its level in the lattice – `BOTTOM` $\sqsubseteq$ `TYPE` $\sqsubseteq$ `CONST_OBJ` = `CONST_LAMBDA` $\sqsubseteq$ `TOP`.

Each cell starts at `TOP`, meaning that it could, in future, take on a constant value. `CONST_OBJ` and `CONST_LAMBDA` occupy the level below, with `.value` pointing to the constant (`robject_t` or `cfunction_t` respectively,) and `.type` to its `rtype_t`. `TYPE` is below these, with a known `.type` but no `.value`. At `BOTTOM`, neither of these properties can be guaranteed.

(the `cellstate` enumeration is reverse order so that `calloc` of the `cells` array initialises the `.state` fields to `TOP`)

⟨*cell_t*⟩≡
```
typedef enum { TOP, CONST_OBJ, CONST_LAMBDA, TYPE, BOTTOM } cellstate;
typedef struct
{
    cellstate state;
    rtype_t *type;
    void *value;
} cell_t;
```

A LatticeCell may be initialised to `TOP` or `BOTTOM` with `cell_init`.

⟨*cell initialisers*⟩≡
```
static inline void cell_init(cell_t *cell, cellstate state)
    { cell->state = state; }
```

A constant value is assigned by `cell_set_const_at`. Note that `obj` could have a type which differs from `type`, but it will always be the case that the former is a subtype of the latter.

⟨*cell initialisers*⟩+≡
```
static inline void cell_set_const_at(cell_t *cell, robject_t *obj, rtype_t *type)
{
    *cell = (cell_t) {
        .state = CONST_OBJ,
        .type = type,
        .value = obj
    };
}
```

If that distinction is unnecessary, `cell_set_const` suffices.

⟨*cell initialisers*⟩+≡
```
static inline void cell_set_const(cell_t *cell, robject_t *obj)
    { cell_set_const_at(cell, obj, r_typeof(obj)); }
```

The value of a `LAMBDA` node is assigned by `cell_set_lambda` – these are tracked separately since they exist only at compile time and are not represented by `robject_ts`. The `.type` of such a cell is determined by its `fn` (Subsection 5.1.1).

⟨*cell initialisers*⟩+≡
```
static inline void cell_set_lambda(cell_t *cell, cfunction_t *fn)
{
    *cell = (cell_t) {
        .state = CONST_LAMBDA,
        .type = fn->cl_type,
        .value = fn
    };
}
```

If all that's known about a `cell` is the `type` of value it's seen to hold, this can be assigned by `cell_set_type`.

⟨*cell initialisers*⟩+≡
```
static inline void cell_set_type(cell_t *cell, rtype_t *type)
{
    *cell = (cell_t) {
        .state = TYPE,
        .type = type
    };
}
```

Accessors query the state of a LatticeCell and extract its properties. `cell_type` returns `NULL` if the cell has no valid `.type`.

⟨*cell accessors*⟩≡
```
static inline rtype_t *cell_type(cell_t *cell)
    { return (cell->state == TOP || cell->state == BOTTOM)
             ? NULL : cell->type; }
```

When one of the `cell_const_obj` and `cell_const_lambda` predicates is `true`, a call to `cell_const` will return the cell's constant `.value`.

⟨*cell accessors*⟩+≡
```
static inline bool cell_const_obj(cell_t *cell)
    { return cell->state == CONST_OBJ; }
static inline bool cell_const_lambda(cell_t *cell)
    { return cell->state == CONST_LAMBDA; }
static inline void *cell_const(cell_t *cell)
    { return cell->value; }
```

10.2.2 Meet

Cells are lowered in the lattice according to the rules for ⊓ described in Wegman and Zadeck (1991, Figure 2), extended with the `TYPE` state. `cell_meet` returns the lattice-theoretic meet of `this` and the `other` LatticeCell.

⟨*cell_meet*⟩≡
```
static inline cell_t cell_meet(cell_t *this, cell_t *other)
{
    ⟨meet⟩
}
```

Since it's a symmetric operator, sorting the inputs reduces the cases necessary to consider.

⟨*meet*⟩≡
```
if(this->state > other->state)
{
    cell_t *tmp = this;
    this = other;
    other = tmp;
}
```

When `this` is `TOP` or `other` is `BOTTOM`, the result is `other`.

⟨*meet*⟩+≡
```
if(this->state == TOP || other->state == BOTTOM)
    return *other;
```

When both `this` and `other` are either `CONST_OBJ` or `CONST_LAMBDA`, and have equal `.value`s, their `.type`s are compared – if those are also equal, the cells are identical; the operation is reflexive, so it doesn't matter which is returned. Otherwise, the result will have the (possibly more general) `r_common_type` (Section 20.5).

⟨*meet*⟩+≡
```
if(other->state < TYPE
   && this->state == other->state
   && this->value == other->value)
{
    if(this->type == other->type)
        return *this;
    return (cell_t) {
        .state = this->state,
        .value = this->value,
        .type = r_common_type(this->type, other->type)
    };
}
```

In all other cases – one or more `TYPE` inputs, or unequal `.values` – the result is a `TYPE` at least as general (i.e. as low in the lattice) as the `.type`s of the inputs.

⟨*meet*⟩+≡
```
return (cell_t) {
    .state = TYPE,
    .type = r_common_type(this->type, other->type)
};
```

`cell_changed` will return `true` when `this` cell differs from the `other`.

⟨*cell_changed*⟩≡
```
static inline bool cell_changed(cell_t *this, cell_t *other)
{
    if(this->state != other->state)
        return true;
    bool chg = false;
    switch(this->state)
    {
    case CONST_OBJ:
    case CONST_LAMBDA:
        chg = (this->value != other->value);
        /* fallthrough */
    case TYPE:
        return chg | (this->type != other->type);
    default:
        return false;
    }
}
```

10.3 Analysis

The SCCP algorithm performs data-flow analysis, proceeding as detailed in Wegman and Zadeck (1991, Subsection 3.4).

⟨*sccp*⟩≡
  ⟨*add_val_work*⟩
  ⟨*add_ctrl_work*⟩
  ⟨*add_cond_work*⟩
  ⟨*add_update_work*⟩
  ⟨*set_flag_for*⟩
  ⟨*visit_phi*⟩
  ⟨*visit_node*⟩
  ⟨*cfunc_sccp*⟩

Analysis begins with the control-flow worklist containing the `.entry` block of the function, and will `run` until both worklists are empty.

⟨*cfunc_sccp*⟩≡
```
static void cfunc_sccp(cfunction_t *fn)
{
    bool run;

    worklist_init(&ctrl_work, fn->nblocks);
    worklist_init(&val_work, fn->nnodes);
    add_ctrl_work(fn->entry);
    do
    {
        run = false;
        ⟨block work⟩
        ⟨node work⟩
    } while(run);
    worklist_fini(&ctrl_work);
    worklist_fini(&val_work);
}
```

Blocks are taken from the control-flow worklist until it's drained. If a `block` hasn't already been visited, its corresponding `flag` is set, signifying that execution can reach that point in the program. Its non-`PHI` nodes are added to the value-flow worklist. If the block has only one successor, execution will reach it unconditionally, so it's added to the control-flow worklist by `add_ctrl_work`.

⟨*block work*⟩≡
```
for(cblock_t *block; worklist_take(&ctrl_work, &block, block->id); run = true)
{
    cnode_t *node;

    if(set_flag_for(block))
        continue;
    list_foreach_entry(&block->cnode_head, node, cnode_list)
        if(node->type != CN_PHI && cnode_yields_value(node))
            add_val_work(node);
    if(alen(&block->succ) == 1)
        add_ctrl_work(aref(&block->succ, 0));
}
```

Nodes are taken from the value-flow worklist so that their LatticeCells may be updated. For a `PHI`, this is done by `visit_phi`; for any other `node`, `visit_node` is only invoked if its block has been flagged as reachable along some execution path. If the `cell`'s value changes as a result, the objects affected are added to the worklists by `add_update_work`.

⟨*node work*⟩≡
```
for(cnode_t *node; worklist_take(&val_work, &node, node->id); run = true)
{
    cell_t new, *cell = cell_for(node);

    if(node->type == CN_PHI)
        visit_phi(node, cell, &new);
    else if(flag_for(node->block))
        visit_node(fn, node, cell, &new);
    else
        continue;
    if(cell_changed(&new, cell))
    {
        *cell = new;
```

```
        add_update_work(node);
    }
}
```

A `node` is visited by evaluating its transfer function on the LatticeCells of its operands. By default, the `new` value is, at `BOTTOM`, lacking constant or type information.

⟨*visit_node*⟩≡

```
static void visit_node(cfunction_t *fn, cnode_t *node, cell_t *cell, cell_t *new)
{
    cell_init(new, BOTTOM);
    opt_transfer(fn, node, new);
}
```

`visit_phi` computes the LatticeCell for the given `PHI` `node`. The `new` value is initialised to `TOP`, since `cell_meet` lowers its inputs. For each argument, if the corresponding predecessor block has its `flag` set, execution can traverse the edge between them, so the `arg`'s LatticeCell is included in the meet.

⟨*visit_phi*⟩≡

```
static void visit_phi(cnode_t *node, cell_t *cell, cell_t *new)
{
    int i;

    cell_init(new, TOP);
    array_foreach(&node->phi.args, i)
    {
        if(flag_for(aref(&node->block->pred, i)))
        {
            cnode_t *arg = aref(&node->phi.args, i);

            *new = cell_meet(new, cell_for(arg));
        }
    }
}
```

10.3.1 Worklists

A `node` is added to the value-flow worklist by the `add_val_work` helper.

⟨*add_val_work*⟩≡

```
static inline void add_val_work(cnode_t *node)
    { worklist_push(&val_work, node, node->id); }
```

This is also invoked on each `PHI` in a `block` when the latter is added to the control-flow worklist by the `add_ctrl_work` helper.

⟨*add_ctrl_work*⟩≡

```
static inline void add_ctrl_work(cblock_t *dest)
{
    cnode_t *node;

    list_foreach_entry(&dest->cnode_head, node, cnode_list)
        if(node->type == CN_PHI)
            add_val_work(node);
        else
            break;
    worklist_push(&ctrl_work, dest, dest->id);
}
```

`add_update_work` is called when a `node`'s LatticeCell is updated. Each `.user` of its value is added to the value-flow worklist; `add_cond_work` is invoked instead if the node is the conditional which controls an `IF`.

⟨*add_update_work*⟩≡
```
static inline void add_update_work(cnode_t *node)
{
    cnode_t *user;

    array_foreach_entry(&node->users, user)
    {
        if(user->type == CN_IF)
            add_cond_work(node, user);
        else if(cnode_yields_value(user))
            add_val_work(user);
    }
}
```

In this case, only one successor of the `IF`'s block may need adding to the control-flow worklist – `tb` if the conditional has the constant `boolean` value `true`, `fb` if it's `false`. Otherwise its execution isn't statically predictable, so both successors are added.

⟨*add_cond_work*⟩≡
```
static void add_cond_work(cnode_t *node, cnode_t *user)
{
    cell_t *cell = cell_for(node);
    cblock_t *block = user->block;
    cblock_t *tb = aref(&block->succ, 0),
             *fb = aref(&block->succ, 1);

    if(cell_const_obj(cell)
       && r_typeof(cell_const(cell)) == r_type_boolean)
    {
        rboolean_t val = UNBOX(rboolean_t, cell_const(cell));
        if(val == true)
        {
            add_ctrl_work(tb);
            return;
        }
        else if(val == false)
        {
            add_ctrl_work(fb);
            return;
        }
    }
    add_ctrl_work(tb);
    add_ctrl_work(fb);
}
```

The `set_flag_for` helper unconditionally sets the corresponding `flag` for the given `block` to `true` and returns the previous value.

⟨*set_flag_for*⟩≡
```
static inline bool set_flag_for(cblock_t *block)
{
    bool visited = flags[block->id];
    flags[block->id] = true;
    return visited;
}
```

10.3.2 Transfer Function

The semantics of the language with respect to the constant and type properties of LatticeCells are given by the *transfer function*.

⟨*transfer*⟩≡
  ⟨*constant_convert*⟩
  ⟨*set_const_global*⟩
  ⟨*set_const_value*⟩
  ⟨*set_const_closed*⟩
  ⟨*transfer_ref*⟩
  ⟨*transfer_copy*⟩
  ⟨*transfer_call*⟩
  ⟨*opt_transfer*⟩

`opt_transfer` assigns to `cell` the LatticeCell representing the value yielded by the `node`, given the LatticeCells of its inputs.

⟨*opt_transfer*⟩≡
```
void opt_transfer(cfunction_t *fn, cnode_t *node, cell_t *cell)
{
    switch(node->type)
    {
        ⟨transfer switch⟩
    default:
        break; /* NOTREACHED */
    }
}
```

The cell for a `LAMBDA` has the constant value of its `.lambda.function`.

⟨*transfer switch*⟩≡
```
case CN_LAMBDA:
    cell_set_lambda(cell, node->lambda.function);
    break;
```

The cell for a `CONST` holds the `.constant` value itself. The `.decl` of the node can differ from `r_typeof(.constant)` when e.g. boxing scalars or checking references.

⟨*transfer switch*⟩+≡
```
case CN_CONST:
    cell_set_const_at(cell, node->constant, node->decl);
    break;
```

An argument-introducing `BIND` without a declared type should stay untyped (so it can be specialised later, after being inlined.)

⟨*transfer switch*⟩+≡
```
case CN_BIND:
    if(!node->set.value && node->decl)
        cell_set_type(cell, node->decl);
    break;
```

The other node types are handled out-of-line.

⟨*transfer switch*⟩+≡
```
case CN_CALL_FAST:
case CN_CALL:
    transfer_call(node, cell);
    break;
case CN_REF:
    transfer_ref(cell, node->ref.var, fn);
    break;
case CN_COPY:
    transfer_copy(cell, node->copy.value, node->decl);
    break;
case CN_BUILTIN:
    trans_builtin(TRANSFER, node, cell, node->builtin.bi,
                  &node->builtin.args);
    break;
```

When the `target` of a `CALL`/`CALL_FAST` node has a callable `type` (Section 21.2), the LatticeCell for the `CALL` is set to the type that the target returns. When the `cell` refers to a known builtin, the `maybe_trans_builtin` helper will invoke its transfer function (if it provides one.)

⟨*transfer_call*⟩≡
```
static void transfer_call(cnode_t *node, cell_t *cell)
{
    cell_t *target = cell_for(node->call.target);
    rtype_t *type = cell_type(target);

    if(!type || !rtype_is_callable(type))
        return;
    cell_set_type(cell, type->sig->ret_type);
    maybe_trans_builtin(TRANSFER, node, cell, target);
}
```

Since SSA conversion (Chapter 9) has lowered most `LEXICAL` variables to value flow, the `REF` transfer function need only concern itself with the remaining cases.

If `var` isn't constant, its declared type (if any) is assigned to its LatticeCell. Otherwise, if it's a `GLOBAL_INT`, its initialising node `.intl.set` provides the `cell`'s constant value. A `GLOBAL_EXT` also has a constant value available, and this is extracted by `set_const_global`. A `LEXICAL` variable, at this point, must be captured by the function's `LAMBDA` from an enclosing lexical ancestor, and its initial value is assigned by `set_const_closed` to the LatticeCell.

⟨*transfer_ref*⟩≡
```
static void transfer_ref(cell_t *cell, cvar_t *var, cfunction_t *fn)
{
    if(!var->is_const)
    {
        if(var->decl)
            cell_set_type(cell, var->decl);
        return;
    }
    switch(var->type)
    {
    case GLOBAL_INT:
        set_const_value(cell, var->intl.set->set.value);
        return;
    case GLOBAL_EXT:
```

```
        set_const_global(cell, var->extl.global);
        return;
    case LEXICAL:
        set_const_closed(cell, var, fn);
        break;
    }
}
```

The `.val` union of a `global` can hold a scalar or an object; in the former case, it must be boxed to form a suitable constant value for the `cell`.

⟨*set_const_global*⟩≡

```
static inline void set_const_global(cell_t *cell, rglobal_t *global)
{
    rtype_t *type = global->decl;
    robject_t *obj = rtype_is_scalar(type)
        ? c_intern(r_box(type, &global->val))
        : global->val.object;

    cell_set_const(cell, obj);
}
```

The `var` at index `i` in the function's `.closure` is initialised, via capture, with the value of the node at the corresponding index in the `LAMBDA`'s `.closure`, so this node also provides the value of the `cell`.

⟨*set_const_closed*⟩≡

```
static inline void set_const_closed(cell_t *cell, cvar_t *var, cfunction_t *fn)
{
    int i = index_of_var(&fn->closure, var);
    set_const_value(cell, aref(&fn->node->lambda.closure, i));
}
```

If the `node` that yields the initial value for a variable is outside the function being analysed, `cell_for` can't be used and `set_const_value` must fall back to direct examination. `CONST` and `LAMBDA` provide appropriate constant values, but anything else (conservatively) gives just a type.

⟨*set_const_value*⟩≡

```
static inline void set_const_value(cell_t *cell, cnode_t *node)
{
    switch(node->type)
    {
    case CN_CONST:
        cell_set_const_at(cell, node->constant, node->decl);
        break;
    case CN_LAMBDA:
        cell_set_lambda(cell, node->lambda.function);
        break;
    default:
        cell_set_type(cell, decl_type(node->decl));
        break;
    }
}
```

The transfer function for a `COPY` node models the semantics of type conversion.

⟨*transfer_copy*⟩≡
```
static inline void transfer_copy(cell_t *cell, cnode_t *src, rtype_t *type)
{
    cell_t *scell = cell_for(src);
    rtype_t *styp = cell_type(scell);

    if(cell_const_obj(scell))
    {
        ⟨copy object⟩
    }
    else if(cell_const_lambda(scell))
    {
        ⟨copy lambda⟩
    }
    ⟨copy type⟩
}
```

If the source node is known to have a constant value, and it can be converted to the requested `type`, the `cell` can be assigned the converted value.

⟨*copy object*⟩≡
```
robject_t *val = cell_const(scell);
bool valid;

val = constant_convert(val, type, &valid);
if(valid)
{
    cell_set_const(cell, val);
    return;
}
```

If the source is known to yield a `LAMBDA`, and its signature is compatible with the requested `type`, the `cell` can be assigned its value.

⟨*copy lambda*⟩≡
```
cfunction_t *fn = cell_const(scell);

if(r_subtypep(fn->cl_type, type))
{
    cell_set_lambda(cell, fn);
    return;
}
```

In case the type `styp` of the source is a subtype of the `type` requested, the `cell` can be assigned the former, as the more precise information can improve optimisation. Otherwise it's assigned the latter; if incompatible, the error will be noted during a later pass.

⟨*copy type*⟩≡
```
if(styp && r_subtypep(styp, type))
    cell_set_type(cell, styp);
else
    cell_set_type(cell, type);
```

`constant_convert` takes a source value `sval` and a destination type `dtype`, returning either the converted object with `pvalid` `true`, or `NULL` with `pvalid` `false`.

⟨*constant_convert*⟩≡
```
robject_t *constant_convert(robject_t *sval, rtype_t *dtype, bool *pvalid)
{
    rtype_t *stype = r_typeof(sval);

    *pvalid = false;
    ⟨identity⟩
    if(rtype_is_scalar(dtype))
    {
        ⟨to scalar⟩
    }
    else if(rtype_is_scalar(stype))
    {
        ⟨from scalar⟩
    }
    else if(r_subtypep(stype, dtype))
    {
        ⟨widening⟩
    }
    return NULL;
}
```

Conversion between the same type is a no-op.

⟨*identity*⟩≡
```
if(stype == dtype)
{
    *pvalid = true;
    return sval;
}
```

If the value is a scalar, it can be converted to any other scalar type.

⟨*to scalar*⟩≡
```
if(rtype_is_scalar(stype))
{
    rvalue_union_t val;

    *pvalid = true;
    scalar_convert(&val, sval, dtype, stype);
    return c_intern(r_box(dtype, &val));
}
```

Converting a scalar value to an `object` requests that it be boxed by the VM, but constants in the compiler are boxed already, so this is also a no-op.

⟨*from scalar*⟩≡
```
if(dtype == r_type_object)
{
    *pvalid = true;
    return sval;
}
```

A value of reference type is also a value of one of its supertypes.

⟨*widening*⟩≡
```
*pvalid = true;
return sval;
```

10.4 Optimisation

Given the LatticeCells and ExecutableFlags discovered by analysis, the program is optimised by transforming its nodes and blocks into a more efficient configuration while maintaining the equivalence of its output.

⟨*optimise*⟩≡
⟨*prune_block*⟩
⟨*prune_dead_blocks*⟩
⟨*cfunc_optimise*⟩

⟨*transform*⟩≡
⟨*call_become_builtin*⟩
⟨*call_become_copy*⟩
⟨*node_become_constant*⟩
⟨*transform_call*⟩
⟨*transform_phi*⟩
⟨*transform_if*⟩
⟨*transform_set*⟩
⟨*shuffle_phi*⟩
⟨*fold_constant*⟩
⟨*update_decl*⟩
⟨*opt_transform*⟩

Each block in the function is visited by `cfunc_optimise`. Those without a `flag` set can't be reached by any path of execution through the program, and may be `pruned` from the control-flow graph.

The nodes in each executable block may be modified by `opt_transform`, up to and including relocation or removal, so the `list_` traversal must be `_safe`.

⟨*cfunc_optimise*⟩≡
```
static cresult cfunc_optimise(cfunction_t *fn)
{
    cblock_array_t pruned;
    cresult res = SUCCESS;
    cblock_t *block;

    array_init(&pruned, 0);
    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        cnode_t *node, *tmp;

        if(!flag_for(block))
        {
            array_push(&pruned, block);
            continue;
        }
        list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
            res |= opt_transform(fn, node, cell_for(node));
    }
    res |= prune_dead_blocks(&pruned);
    array_fini(&pruned);
    return res;
}
```

The blocks found to be unreachable are removed by `prune_dead_blocks`. `prune_block` is invoked on each; it unconditionally examines the `.ids` of predecessors and successors, so we must defer the calls to `cblock_free`.

⟨*prune_dead_blocks*⟩≡
```
static cresult prune_dead_blocks(cblock_array_t *arr)
{
    cblock_t *block;

    if(!opt.opt_dce || alen(arr) == 0)
        return SUCCESS;
    array_foreach_entry(arr, block)
        prune_block(block);
    array_foreach_entry(arr, block)
        cblock_free(block);
    return CHANGED;
}
```

Pruning the unreachable `block` involves breaking the links between itself and each of its reachable predecessors and successors (since unreachable ones will also be removed, they can be ignored here.) All its nodes must cease using any others, as `cblock_free` calls `cnode_free`, not `cnode_remove`.

⟨*prune_block*⟩≡
```
static void prune_block(cblock_t *block)
{
    cblock_t *other;
    cnode_t *node;

    array_foreach_entry(&block->pred, other)
        if(flag_for(other))
            remove_link(&other->succ, block);
    array_foreach_entry(&block->succ, other)
        if(flag_for(other))
            remove_link(&other->pred, block);
    list_foreach_entry(&block->cnode_head, node, cnode_list)
        cnode_unuse_all(node);
}
```

10.4.1 Transformation

By analogy with `opt_transfer`, the transformation function `opt_transform` modifies the `node` in light of the (potentially new) information given by the `cell`. If anything is `CHANGED` as a result, the outer analysis-optimisation loop will run again on the updated IR.

⟨*opt_transform*⟩≡
```
cresult opt_transform(cfunction_t *fn, cnode_t *node, cell_t *cell)
{
    cresult res = update_decl(node, cell);

    switch(node->type)
    {
        ⟨transform switch⟩
    default:
        break;
    }
    return res;
}
```

`REF`, `COPY` and `BUILTIN` nodes have no specific transformations but, since they yield values, they may become `CONST`s via `fold_constant`.

⟨*transform switch*⟩≡
```
case CN_REF:
case CN_COPY:
case CN_BUILTIN:
    if(fold_constant(node, cell))
        res |= CHANGED;
    break;
```

`CALL`/`CALL_FAST` nodes may also be folded as constants but, if that isn't possible, may undergo further transformation via `transform_call`.

⟨*transform switch*⟩+≡
```
case CN_CALL:
case CN_CALL_FAST:
    if(fold_constant(node, cell))
        res |= CHANGED;
    else
        res |= transform_call(node, cell);
    break;
```

In a similar manner, `PHI` nodes may be folded, or transformed by `transform_phi`.

⟨*transform switch*⟩+≡
```
case CN_PHI:
    if(fold_constant(node, cell))
        res |= CHANGED;
    else
        res |= transform_phi(node);
    break;
```

`SET` and `IF` nodes don't yield values, but have transformation functions of their own.

⟨*transform switch*⟩+≡
```
case CN_SET:
    res |= transform_set(node, cell);
    break;
case CN_IF:
    res |= transform_if(node);
    break;
```

If the analysis has computed a different type for the `node` (assuming it yields a value,) its `.decl` field is updated from its corresponding `cell`, and the appropriate result flag raised.

⟨*update_decl*⟩≡
```
static cresult update_decl(cnode_t *node, cell_t *cell)
{
    if(cnode_yields_value(node))
    {
        rtype_t *type = cell_type(cell);

        if(type && node->decl != type)
        {
            node->decl = type;
            return CHANGED;
        }
    }
    return SUCCESS;
}
```

When the `target` of a `CALL` or `CALL_FAST` node is known to be a `LAMBDA`, the `node` is added to the set of potential `inlines`. When it's known to be a builtin, `maybe_trans_builtin` will invoke the builtin's transformation function, returning the result flags.

⟨*transform_call*⟩≡
```
static cresult transform_call(cnode_t *node, cell_t *cell)
{
    cell_t *target = cell_for(node->call.target);

    if(opt.opt_inline && cell_const_lambda(target))
    {
        array_push(&inlines, node);
        return SUCCESS;
    }
    return maybe_trans_builtin(TRANSFORM, node, cell, target);
}
```

A `PHI` node's arguments correspond to inedges from predecessors of the `block` in which it resides. If any of the latter will be removed by `prune_dead_blocks`, the former must be as well. The node unregisters itself from using them; the others are copied to the scratch buffer `buf`.

Should this leave the `node` with only one argument, it's redundant – the argument takes the place of the `PHI`, which can then be `cnode_remove`d.

Otherwise, if the number of arguments has changed, the node's `.phi.args` array is emptied and refilled with the non-`NULL` elements from the scratch buffer.

⟨*transform_phi*⟩≡
```
static cresult transform_phi(cnode_t *node)
{
    cblock_t *block = node->block;
    cnode_t *buf[alen(&block->pred)];
    bool purge = true;
    int i, j = 0;

    if(!opt.opt_dce)
        return SUCCESS;
    array_foreach(&block->pred, i)
    {
        cblock_t *pred = aref(&block->pred, i);
        cnode_t *arg = aref(&node->phi.args, i);

        if(flag_for(pred))
        {
            if(j > 0 && arg != buf[j-1])
                purge = false;
            buf[j++] = arg;
        }
        else
            cnode_remove_user(node, arg);
    }
    if(purge)
    {
        cnode_replace_in_users(node, buf[0]);
        cnode_remove(node);
        return CHANGED;
    }
    else if(j == alen(&block->pred))
        return SUCCESS;
```

```
        array_clear(&node->phi.args);
        for(i=0; i<j; i++)
            array_push(&node->phi.args, buf[i]);
        return CHANGED;
    }
```

When one or both successors of an `IF` node's `block` are going to be removed, the `node` is redundant and may be removed also.

⟨*transform_if*⟩≡

```
    static cresult transform_if(cnode_t *node)
    {
        cblock_t *block = node->block;
        cblock_t *succ;

        if(!opt.opt_dce)
            return SUCCESS;
        array_foreach_entry(&block->succ, succ)
        {
            if(!flag_for(succ))
            {
                cnode_remove(node);
                return CHANGED;
            }
        }
        return SUCCESS;
    }
```

As a convenience, a constant `GLOBAL_INT` variable takes on the recovered type of the value assigned by its initialising `SET` node.

⟨*transform_set*⟩≡

```
    static cresult transform_set(cnode_t *node, cell_t *cell)
    {
        cvar_t *var = node->set.var;

        if(var->type == GLOBAL_INT && var->is_const)
        {
            cell_t *value = cell_for(node->set.value);
            rtype_t *type = cell_type(value);

            if(type && var->decl != type)
            {
                var->decl = type;
                return CHANGED;
            }
        }
        return SUCCESS;
    }
```

If its corresponding `cell` is determined to have constant value, `fold_constant` transforms the `node` into a `CONST`.

⟨*fold_constant*⟩≡

```
static bool fold_constant(cnode_t *node, cell_t *cell)
{
    if(opt.opt_constfold && cell_const_obj(cell))
    {
        robject_t *val = cell_const(cell);

        if(node->type == CN_PHI)
            shuffle_phi(node);
        node_become_constant(node, val);
        return true;
    }
    return false;
}
```

Since `PHI` nodes appear before the others in a block, the `node` being folded is shuffled forward, beyond its former fellows (it'll be encountered again during the list-order traversal in `cfunc_optimise`, but this has no effect since `CONST`s are ignored by `opt_transform`.)

⟨*shuffle_phi*⟩≡

```
static inline void shuffle_phi(cnode_t *node)
{
    list_t *link;

    for(link = node->cnode_list.next;
        link != &node->block->cnode_head;
        link = link->next)
    {
        cnode_t *next = list_entry(link, node, cnode_list);

        if(next->type != CN_PHI)
            break;
    }
    list_remove(&node->cnode_list);
    list_add_before(link, &node->cnode_list);
}
```

`node_become_constant` invokes `cnode_reset` before `node` takes on its new guise as a `CONST` with the given `value`. As noted, its `type` may be different to the `r_typeof` the latter.

⟨*node_become_constant*⟩≡

```
void node_become_constant(cnode_t *node, robject_t *val)
{
    rtype_t *type = node->decl;

    cnode_reset(node);
    node->type = CN_CONST;
    node->decl = type;
    node->constant = val;
}
```

10.5 Builtins

Builtin functions may supply optional callbacks which the compiler will invoke during analysis and optimisation to enact their unique semantics.

⟨*builtins*⟩≡
  ⟨*builtin predicates*⟩
  ⟨*trans_builtin*⟩
  ⟨*maybe_trans_builtin*⟩

These are specified by the `builtin_ops_t` structure referenced by the `cbuiltin_t` associated with the function (Section 21.6).

The `.trans_fn` callback is invoked first as a `TRANSFER` function to provide a value for the `cell`; then to `TRANSFORM` the node from a `CALL` to a `BUILTIN` if possible. The optional `.enforce_fn` will be called during postpass (Chapter 13) to convert the node's inputs to an appropriate type; `.generate_fn` is responsible for emitting the appropriate bytecode instruction during code generation (Section 15.5).

`.is_pure` and `.is_void` are wrapped by predicates, since `builtin_ops_t` is private to this subsystem.

⟨*builtin_ops_t*⟩≡
```
typedef enum { TRANSFER, TRANSFORM } opt_phase;
typedef struct builtin_ops
{
    cresult (*trans_fn)(opt_phase, cnode_t *, cell_t *,
                        const cbuiltin_t *, cnode_array_t *);
    cresult (*enforce_fn)(cnode_t *);
    void (*generate_fn)(cnode_t *);
    bool (*is_pure)(cnode_t *, bool);
    bool is_void;
} builtin_ops_t;
```

`builtin_is_void` when the builtin returns no meaningful value; in this case it won't require a location to be allocated for its result.

`builtin_is_pure` if the call to builtin `bi` at `node`, given `may_alias`, can be considered to be functionally pure (Subsection 5.2.4, `cnode_is_pure`).

⟨*builtin predicates*⟩≡
```
bool builtin_is_pure(const cbuiltin_t *bi, cnode_t *node, bool may_alias)
    { return bi->ops && bi->ops->is_pure
             && bi->ops->is_pure(node, may_alias); }
bool builtin_is_void(const cbuiltin_t *bi, cnode_t *node)
    { return bi->ops && bi->ops->is_void; }
```

`trans_builtin` is responsible for invoking the `.trans_fn` callback when the builtin `bi` provides one. It's passed the `CALL` or `BUILTIN` `node`, its `cell`, and the array `args` containing its arguments.

⟨*trans_builtin*⟩≡
```
static cresult trans_builtin(opt_phase phase, cnode_t *node, cell_t *cell,
                             const cbuiltin_t *bi, cnode_array_t *args)
{
    if(bi && bi->ops && bi->ops->trans_fn)
        return bi->ops->trans_fn(phase, node, cell, bi, args);
    return SUCCESS;
}
```

The maybe_trans_builtin function calls trans_builtin when the node is a CALL of a constant builtin target. Since this examines the .call.args array given by the user, it isn't applicable to CALL_FAST nodes.

⟨*maybe_trans_builtin*⟩≡
```
static cresult maybe_trans_builtin(opt_phase phase, cnode_t *node,
                                   cell_t *cell, cell_t *target)
{
    if(opt.opt_builtin && cell_const_obj(target)
       && node->type == CN_CALL)
    {
        robject_t *obj = cell_const(target);
        if(rtype_is_callable(r_typeof(obj)))
        {
            rcallable_t *cl = (rcallable_t *)obj;
            if(rcall_is_builtin(cl))
            {
                rbuiltin_t *rbi = (rbuiltin_t *)obj;
                return trans_builtin(phase, node, cell, rbi->cbi,
                                     &node->call.args);
            }
        }
    }
    return SUCCESS;
}
```

During the TRANSFORM phase, .trans_fn may decide the functionality of the node in question is best implemented by a bytecode instruction. call_become_builtin will be invoked to change the node's .type and initialise the .builtin.bi and .builtin.optype fields (the array of args is taken before cnode_reset to avoid needing to undo the effects of its call to cnode_unuse_all).

⟨*call_become_builtin*⟩≡
```
void call_become_builtin(cnode_t *node, const cbuiltin_t *bi,
                         rtype_t *optype, rtype_t *type)
{
    cnode_array_t args = node->call.args;

    node->call.args = (cnode_array_t) ARRAY_INIT;
    cnode_reset(node);
    node->type = CN_BUILTIN;
    node->decl = type;
    node->builtin.bi = bi;
    node->builtin.args = args;
    node->builtin.optype = optype;
}
```

10.5.1 Argument Helpers

arg_type returns the recovered type of the argument with index n in the args array. Omitted arguments have the nil type, which may be inappropriate; caveat user.

⟨*arg_type*⟩≡
```
static inline rtype_t *arg_type(cnode_array_t *args, int n)
{
    cnode_t *arg = aref(args, n);
    return arg ? cell_type(cell_for(arg)) : r_type_nil;
}
```

`check_arg_names` returns `true` when a `CALL` node does not have any named arguments.

⟨*check_arg_names*⟩≡
```
static inline bool check_arg_names(cnode_t *node)
{
    if(node->type == CN_CALL && alen(&node->call.names) > 0)
        return false;
    return true;
}
```

`check_args` returns `true` when there are no missing arguments and, if the node is a `CALL`, no argument names are present.

⟨*check_args*⟩≡
```
static inline bool check_args(cnode_t *node, cnode_array_t *args)
{
    cnode_t *arg;
    array_foreach_entry(args, arg)
        if(!arg)
            return false;
    return check_arg_names(node);
}
```

`check_nargs` additionally ensures that the call has the specified `arity`.

⟨*check_nargs*⟩≡
```
static inline bool check_nargs(int arity, cnode_t *node, cnode_array_t *args)
{
    if(alen(args) != arity)
        return false;
    return check_args(node, args);
}
```

10.6 Miscellanea

⟨*opt.h*⟩≡
```
⟨cell_t⟩
⟨cell accessors⟩
⟨cell initialisers⟩
extern cell_t *cells;
extern bool *flags;
⟨global helpers⟩
extern cnode_array_t inlines;
⟨builtin_ops_t⟩
// for arith/opt.c
void call_become_builtin(cnode_t *node, const cbuiltin_t *bi,
                         rtype_t *optype, rtype_t *type);
void call_become_copy(cnode_t *node, rtype_t *type);
// for opt_post.c
robject_t *constant_convert(robject_t *sval, rtype_t *dtype, bool *pvalid);
void node_become_constant(cnode_t *node, robject_t *val);
// for opt_sccp.c (entry points)
void opt_transfer(cfunction_t *fn, cnode_t *node, cell_t *cell);
cresult opt_transform(cfunction_t *fn, cnode_t *node, cell_t *cell);
// opt_post.c
cresult call_normalise(cnode_t *node, funsig_t *sig);
// opt_inline.c
cresult inline_lambdas(cfunction_t *outer, cnode_array_t *nodes);
⟨arg_type⟩
```

⟨*check_arg_names*⟩
⟨*check_args*⟩
⟨*check_nargs*⟩

⟨*sccp includes*⟩≡
```
#include "global.h"
#include "worklist.h"
#include "ir.h"
#include "opt.h"
```

⟨*trans includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "opt.h"
bool scalar_convert(void *dest, robject_t *obj, rtype_t *to, rtype_t *from);
```

Chapter 11

Inline Expansion

If a call site is known to refer to another function in the program being compiled, it's possible to *inline* a copy of the called function's body into the caller at that location. This transformation is also known as "procedure integration" or β-expansion.

The inlining process commingles the contexts of the call site and called function (Wegman and Zadeck, 1991, Section 6) which can allow further optimisation – in particular, the specialisation of functions with generic behaviour.

⟨*opt_inline.c*⟩≡
⟨*includes*⟩
⟨*ir copy*⟩
⟨*ir fixup*⟩
⟨*inline expansion*⟩
⟨*heuristics*⟩
⟨*can_inline*⟩
⟨*inline_lambdas*⟩

11.1 Entry Point

The `inline_lambdas` function is called from the analysis-optimisation loop (Section 10.1) with an array of `nodes`. These contain the call sites within the `caller` function known to invoke `LAMBDA`s whose IR objects are available to the compiler.

Each such `CALL node` is considered; the `callee` extracted from its `.target`'s LatticeCell (Subsection 10.2.1). If `can_inline` allows it, `call_normalise` statically matches the arguments (Section 13.4), and `should_inline` confirms it, the body of the called function is copied and spliced into the caller by `expand_inline`.

⟨*inline_lambdas*⟩≡
```
cresult inline_lambdas(cfunction_t *caller, cnode_array_t *nodes)
{
    cresult res = SUCCESS;
    cnode_t *node;

    if(!opt.opt_inline)
        return SUCCESS;
    array_foreach_entry(nodes, node)
    {
        assert(cell_const_lambda(cell_for(node->call.target)));
        cfunction_t *callee = cell_const(cell_for(node->call.target));
        funsig_t *sig = callee->cl_type->sig;

        if(can_inline(node, caller, callee)
           && call_normalise(node, sig) == CHANGED
```

```
            && should_inline(node, caller, callee))
             res |= expand_inline(node, caller, callee);
    }
    return res;
}
```

Certain conditions prevent the inlining of the `callee` function in its `caller`, and are detected by `can_inline`.

- Functions are not inlined into themselves to avoid excessive expansion.
- Functions are not inlined into the top-level function.
- Functions with closed variables are only inlined into their lexical parent.

These are all overly conservative, and should be relaxed with appropriate implementation changes.

⟨*can_inline*⟩≡

```
static bool can_inline(cnode_t *site, cfunction_t *caller, cfunction_t *callee)
{
    if(caller == callee)
        return false;
    if(!caller->parent)
       return false;
    if(!cfunc_has_closure(callee))
        return true;
    if(callee->parent == caller)
        return true;
    return false;
}
```

11.2 Heuristics

⟨*heuristics*⟩≡
⟨*args_spec*⟩
⟨*args_const*⟩
⟨*score_function*⟩
⟨*should_inline*⟩

A very simple-minded, aggressive heuristic is implemented by the `should_inline` function. The `caller` may have no more than `MAX_NODES` added through inlining. If the `callee` is only called once, at the call `site`, it will be inlined; likewise if it's "small enough" (less than `MIN_NODES` in size, with no child functions of its own). Otherwise, inlining occurs if `score_function` returns a value below the `CUTOFF`.

Each inline expansion decision makes a tradeoff between code size and (ideally) execution time. Needless to say, the effects of the heuristic parameters on various representative codes are significant, and require further investigation.

⟨*should_inline*⟩≡
```
#define MAX_NODES 256
#define MIN_NODES 8
#define CUTOFF 32
static bool should_inline(cnode_t *site, cfunction_t *caller,
                          cfunction_t *callee)
{
    if(caller->ninlined + callee->nnodes > MAX_NODES)
        return false;
    if(cnode_only_used_by(callee->node, site))
        return true;
    if(callee->nfuncs == 0 && callee->nnodes < MIN_NODES)
        return true;
    return score_function(site, callee) < CUTOFF;
}
```

`score_function` computes a weighted sum of the number of nodes and child functions in the `callee`, reduced proportionally by a factor representing the degree to which the actual arguments at the call `site` are specialised or constant.

⟨*score_function*⟩≡
```
#define WEIGHT_FUNC 8
#define WEIGHT_NODE 1
#define WEIGHT_SPEC 0.5
#define WEIGHT_CONST 0.25
static float score_function(cnode_t *site, cfunction_t *fn)
{
    float factor, score;

    score = fn->nfuncs * WEIGHT_FUNC + fn->nnodes * WEIGHT_NODE;
    factor = (args_spec(&site->call.args, fn->cl_type->sig) * WEIGHT_SPEC +
              args_const(&site->call.args) * WEIGHT_CONST);
    score -= score * factor;
    return score;
}
```

`args_spec` returns the proportion of the `args` which are more specialised than (i.e. are strict subtypes of) the declared types in the function signature `sig`.

⟨*args_spec*⟩≡
```
static float args_spec(cnode_array_t *args, funsig_t *sig)
{
    float factor = 0;
    int i, n = alen(args) > 0 ? alen(args) : 1;
    assert(alen(args) == sig->nargs);

    array_foreach(args, i)
    {
        cnode_t *arg = aref(args, i);
        rtype_t *atyp = decl_type(arg ? arg->decl : NULL);
        rtype_t *ftyp = sig->args[i].type;

        if(atyp != ftyp && r_subtypep(atyp, ftyp))
            factor += 1;
    }
    return factor / n;
}
```

`args_const` returns the proportion of the `args` which are constant values (i.e. supplied by `CONST` nodes.)

⟨*args_const*⟩≡
```
static float args_const(cnode_array_t *args)
{
    float factor = 0;
    int i, n = alen(args) > 0 ? alen(args) : 1;

    array_foreach(args, i)
    {
        cnode_t *arg = aref(args, i);

        if(arg && arg->type == CN_CONST)
            factor += 1;
    }
    return factor / n;
}
```

11.3 Inline Expansion

⟨*inline expansion*⟩≡
⟨*nil_actual*⟩
⟨*enforce_inline_args*⟩
⟨*bind_closed_values*⟩
⟨*bind_args*⟩
⟨*link_exits*⟩
⟨*expand_inline*⟩

`expand_inline` is responsible for completing the integration of the `callee` into the body of the `caller`. The IR objects of the former are duplicated and added to the latter via `copy_into`; a pointer is retained to the `entry` block of the copied function. Its `RETURN` nodes are collected in the `exits` array.

The actual arguments of a `CALL` node can have type conversions added during postpass (Section 13.2); this one is being elided, so `enforce_inline_args` is called instead.

The block in the `caller` containing the call `site` is split into two halves, `above` and `below`. Value flow is connected first: if the `callee` has closed variables, their values are gathered `above` the split by `bind_closed_values`. Formal arguments in the copied `entry` block are bound to their actual values by `bind_args`.

Next is control flow: the block `above` the call site is linked to the `entry`. The collected `exits` are connected `below` it by `link_exits`, which returns the `value` yielded by the inlined function. The replacement and removal of the call site completes the inlining process.

⟨*expand_inline*⟩≡
```
static cresult expand_inline(cnode_t *site, cfunction_t *caller,
                             cfunction_t *callee)
{
    cresult res = SUCCESS;
    cnode_array_t exits = ARRAY_INIT;
    cblock_t *entry, *above, *below;
    cnode_t *value;

    array_init(&exits, 1);
    entry = copy_into(caller, callee, NULL, &exits);
    caller->ninlined += callee->nnodes;
    res |= enforce_inline_args(site, callee);
    below = site->block;
    above = split_block(caller, site);
    if(cfunc_has_closure(callee))
        bind_closed_values(above, caller, callee);
    bind_args(site, entry, caller, callee);
    link_blocks(entry, above);
    res |= link_exits(below, callee, &exits, &value);
    cnode_replace_in_users(site, value);
    cnode_remove(site);
    return res | CHANGED;
}
```

The type conversions available to `enforce_inline_args` are more lenient than those required at a `CALL` or `CALL_FAST` node – now unconstrained by the calling convention (Chapter 22), `enforce_decl` need not box scalar actual arguments when their corresponding formal argument lacks a declared type (Subsection 5.2.7).

A `NULL` in the `args` array signifies an omitted actual argument. Its value is provided explicitly by `nil_actual`.

⟨*enforce_inline_args*⟩≡
```
static cresult enforce_inline_args(cnode_t *site, cfunction_t *fn)
{
    cnode_array_t *args = &site->call.args;
    funsig_t *sig = fn->cl_type->sig;
    cresult res = SUCCESS;
    int i;

    array_foreach(args, i)
    {
```

```
        cnode_t **ptr = aptr(args, i);
        rtype_t *adecl = sig->args[i].type;

        if(!*ptr)
            *ptr = nil_actual(site, adecl);
        else
            res |= enforce_decl(site, adecl, ptr, false);
    }
    assert(cnode_compat(site, sig->ret_type) == YES);
    return res;
}
```

During a call, an omitted argument will be given an appropriate zero value by the VM's calling mechanism. In IR, this can be given by a `CONST` node yielding the `nil_init` value of appropriate type `adecl` (the call `site` becomes a user of the new `node`).

⟨*nil_actual*⟩≡

```
static inline cnode_t *nil_actual(cnode_t *site, rtype_t *adecl)
{
    cnode_t *node = cnode_insert_before(site, CN_CONST);
    node->constant = c_intern(nil_init(adecl));
    node->decl = adecl;
    cnode_add_user(site, node);
    return node;
}
```

A reference to a closed variable (Section 7.6) reads its value from the function's captured environment. After the `callee` is inlined, this may no longer be available. The `bind_closed_values` function supplies such references with the values captured by the `LAMBDA` node.

For each constant `var` not bound by a lexical ancestor, a `BIND` node is created in the preceding `block`, assigning to the variable the corresponding `val` from the `.lambda.closure` array.

Other closed variables are either available in the `caller`'s environment (if bound by an ancestor) or don't require treatment here (if not constant).

⟨*bind_closed_values*⟩≡

```
static void bind_closed_values(cblock_t *block, cfunction_t *caller,
                               cfunction_t *callee)
{
    cvar_array_t *vars = &callee->closure;
    cnode_array_t *vals = &callee->node->lambda.closure;
    int i;

    array_foreach(vars, i)
    {
        cvar_t *var = aref(vars, i);
        cnode_t *val = aref(vals, i), *node;

        if(!val)
            continue;
        if(var->local.binder != caller)
            continue;
        node = cnode_append(block, CN_BIND);
        node->decl = val->decl;
        node->set.var = var;
        node->set.value = val;
        cnode_add_user(node, val);
    }
```

```
}
```

Formal arguments receive their actual values in a similar manner in `bind_args`. The `BIND` nodes in the copied entry `block` of the `callee` are examined in order.

The first will have `.set.var` `NULL`, and conveys the `argbits`. Having been computed and stored in the `site`'s `.call.argbits` field by the prior invocation of `call_normalise`, a `CONST` node with this value can replace the `BIND`.

Others (with `.set.value`s `NULL`) reference the formal arguments, in order. Each is assigned the corresponding actual value from the `.call.args` array; this value also replaces the `BIND` node in its users (the latter will be removed during SSA conversion).

⟨*bind_args*⟩≡

```
static void bind_args(cnode_t *site, cblock_t *block, cfunction_t *caller,
                      cfunction_t *callee)
{
    cnode_array_t *args = &site->call.args;
    cnode_t *node;
    int nargs = 0;

    list_foreach_entry(&block->cnode_head, node, cnode_list)
    {
        if(node->type != CN_BIND)
            continue;
        if(!node->set.var)
        {
            cnode_reset(node);
            node->type = CN_CONST;
            node->constant = c_intern(r_box(r_type_int, &site->call.argbits));
            node->decl = r_type_int;
        }
        else if(!node->set.value)
        {
            cnode_t *val = aref(args, nargs);

            node->set.value = val;
            cnode_add_user(node, val);
            cnode_replace_in_users(node, val);
            nargs++;
        }
    }
}
```

The `exits` array passed to `link_exits` contains all of the `RETURN` nodes in the copied `callee`.

Each value returned can have a type conversion added during postpass, similar to the actual arguments of a call. Since the `RETURN` is also being elided, `enforce_decl` is invoked similarly, without the necessity of boxing scalar values in the absence of declared `ret_type`.

The block ended by the `RETURN` is linked to the `block` following the call site; the element in `exits` is replaced by the returned value, and the `RETURN` node is removed.

If the function has more than one `RETURN`, a `PHI` node joins their values (taking ownership of `exits`). Otherwise the array is superfluous, and the single value is returned in `*pval`.

⟨*link_exits*⟩≡

```
static cresult link_exits(cblock_t *block, cfunction_t *callee,
                          cnode_array_t *exits, cnode_t **pval)
{
```

```
    rtype_t *ret_type = callee->cl_type->sig->ret_type;
    cnode_t **ptr;
    cresult res = SUCCESS;

    assert(alen(exits) > 0);
    array_foreach_ptr(exits, ptr)
    {
        cnode_t *node = *ptr;
        assert(node->type == CN_RETURN);
        res |= enforce_decl(node, ret_type, &node->ret.value, false);
        *ptr = node->ret.value;
        link_blocks(block, node->block);
        cnode_remove(node);
    }
    if(alen(exits) > 1)
    {
        cnode_t *arg;

        *pval = cnode_prepend(block, CN_PHI);
        (*pval)->phi.args = *exits;
        array_foreach_entry(exits, arg)
            cnode_add_user(*pval, arg);
    }
    else
    {
        *pval = aref(exits, 0);
        array_fini(exits);
    }
    return res;
}
```

11.4 IR Copy

⟨*ir copy*⟩≡
  ⟨*copy_ctx*⟩
  ⟨*copy_one_function*⟩
  ⟨*copy_functions*⟩
  ⟨*copy_one_var*⟩
  ⟨*number_vars*⟩
  ⟨*copy_vars*⟩
  ⟨*copy_one_block*⟩
  ⟨*copy_one_node*⟩
  ⟨*copy_body*⟩
  ⟨*copy_into*⟩

The process of inlining involves the exact duplication of IR objects. Unfortunately, these make liberal use of pointers to each other. Fortunately, each kind of object is densely numbered and accurately counted. Originals are mapped to copies through an intermediate set of arrays indexed by object `.id`, stored in the `copy_ctx_t` structure.

Functions are copied recursively – if the function `.src` being copied is a descendent of the function being inlined, `.parent` is non-`NULL` and points to the context established for its enclosing function.

The remaining fields store the functions, blocks, nodes and variables copied from the function `.src`.

⟨*copy_ctx*⟩≡
```
typedef struct copy_ctx
{
    struct copy_ctx *parent;
    cfunction_t *src;
    cfunction_t **functions;
    cblock_t **blocks;
    cnode_t **nodes;
    cvar_t **vars;
} copy_ctx_t;
```

The `copy_into` function copies all IR objects contained in the function `src` into the function `dest`. Variables are copied (by `copy_vars`) before nested functions (by `copy_functions`). `copy_body` copies blocks and nodes, then `fixup` rewrites intra-IR pointers. Duplication complete, the context is emptied and the copied `entry` block is returned.

⟨*copy_into*⟩≡
```
static void fixup(copy_ctx_t *ctx, cfunction_t *dest, cnode_array_t *exits);
static cblock_t *copy_into(cfunction_t *dest, cfunction_t *src,
                           copy_ctx_t *parent, cnode_array_t *exits)
{
    cblock_t *entry;
    copy_ctx_t ctx = { .src = src, .parent = parent };

    copy_vars(&ctx, dest);
    copy_functions(&ctx, dest);
    entry = copy_body(&ctx, dest);
    fixup(&ctx, dest, exits);
    if(ctx.vars)
        xfree(ctx.vars);
    if(ctx.nodes)
        xfree(ctx.nodes);
    if(ctx.blocks)
        xfree(ctx.blocks);
    if(ctx.functions)
        xfree(ctx.functions);
    return entry;
}
```

11.4.1 Variables

Variables aren't numbered by the ubiquitous `cfunc_rdfo`, so `copy_vars` must first count and number them with `number_vars`. Assuming the function `src` contains any, the `.vars` array is initialised and populated with a copy of each `var` on the function's `cvar_head` list.

⟨*copy_vars*⟩≡
```
static void copy_vars(copy_ctx_t *ctx, cfunction_t *dest)
{
    cfunction_t *src = ctx->src;
    int nvars = number_vars(src);
    cvar_t *var;

    if(nvars == 0)
        return;
    ctx->vars = xmalloc(nvars * sizeof(cvar_t *));
    list_foreach_entry(&src->cvar_head, var, cvar_list)
        ctx->vars[var->id] = copy_one_var(dest, var);
}
```

`number_vars` returns the number of variables owned by `fn`, after assigning them `.ids` in list order.

⟨*number_vars*⟩≡
```
static int number_vars(cfunction_t *fn)
{
    int nvars = 0;
    cvar_t *var;

    list_foreach_entry(&fn->cvar_head, var, cvar_list)
        var->id = nvars++;
    return nvars;
}
```

`copy_one_var` copies the `LEXICAL` variable `var` to the function `dest`, updating its `.local.binder` accordingly.

⟨*copy_one_var*⟩≡
```
static cvar_t *copy_one_var(cfunction_t *dest, cvar_t *var)
{
    assert(!cvar_is_global(var));
    cvar_t *copy = cvar_create(var->name, var->type, var->decl);

    copy->local = var->local;
    copy->local.binder = dest;
    copy->is_const = var->is_const;
    copy->id = var->id;
    list_add_before(&dest->cvar_head, &copy->cvar_list);
    return copy;
}
```

11.4.2 Functions

The child functions of the function `src`, if any, are copied into `dest` by `copy_functions`; these copies populate the context `.functions` array.

⟨*copy_functions*⟩≡
```
static void copy_functions(copy_ctx_t *ctx, cfunction_t *dest)
{
    cfunction_t *src = ctx->src, *fn;

    if(src->nfuncs == 0)
        return;
    ctx->functions = xmalloc(src->nfuncs * sizeof(cfunction_t *));
    list_foreach_entry(&src->cfunc_head, fn, cfunc_list)
        ctx->functions[fn->id] = copy_one_function(ctx, dest, fn);
}
```

For each, `copy_one_function` creates a new child of the `dest` function, initialises its fields, and recursively invokes `copy_into` to duplicate the source `child`'s objects (its `.node` is set when we fix up the copied `LAMBDA`; `.closure` is also set in fixup.)

⟨*copy_one_function*⟩≡
```
static cblock_t *copy_into(cfunction_t *, cfunction_t *,
                           copy_ctx_t *, cnode_array_t *);
static cfunction_t *copy_one_function(copy_ctx_t *ctx, cfunction_t *dest,
                                      cfunction_t *child)
{
    cfunction_t *copy = cfunc_create(dest);

    copy->cl_type = child->cl_type;
    copy->nfuncs = child->nfuncs;
    copy->nblocks = child->nblocks;
    copy->nnodes = child->nnodes;
    copy->id = child->id;
    copy->ninlined = child->ninlined;
    copy->entry = copy_into(copy, child, ctx, NULL);
    return copy;
}
```

11.4.3 Blocks & Nodes

The nodes and blocks of `src` are copied into `dest` by `copy_body`, and the context arrays populated. The copy made of the `entry` block is returned.

⟨*copy_body*⟩≡
```
static cblock_t *copy_body(copy_ctx_t *ctx, cfunction_t *dest)
{
    cfunction_t *src = ctx->src;
    cblock_t *block, *entry = NULL;

    assert(src->nblocks>0 && src->nnodes>0);
    ctx->blocks = xmalloc(src->nblocks * sizeof(cblock_t *));
    ctx->nodes = xmalloc(src->nnodes * sizeof(cnode_t *));
    list_foreach_entry(&src->cblock_head, block, cblock_list)
    {
        cnode_t *node;
        cblock_t *newblock = copy_one_block(dest, block);

        ctx->blocks[block->id] = newblock;
```

```
            if(block == src->entry)
                entry = newblock;
            list_foreach_entry(&block->cnode_head, node, cnode_list)
            {
                cnode_t *newnode = copy_one_node(newblock, node);

                ctx->nodes[node->id] = newnode;
            }
        }
        return entry;
    }
```

`copy_one_block` creates a copy of a `block` and links it into the `dest` function's `.cblock_head` list; its `.succ` and `.pred` arrays will be copied during fixup.

⟨*copy_one_block*⟩≡

```
    static cblock_t *copy_one_block(cfunction_t *dest, cblock_t *block)
    {
        cblock_t *copy = cblock_create();

        copy->id = block->id;
        copy->start = block->start;
        copy->end = block->end;
        list_add_before(&dest->cblock_head, &copy->cblock_list);
        return copy;
    }
```

`copy_one_node` copies a `node` into the given `block`; since most of its fields are pointers to other nodes, they will be copied during fixup.

⟨*copy_one_node*⟩≡

```
    static cnode_t *copy_one_node(cblock_t *block, cnode_t *node)
    {
        cnode_t *copy = cnode_append(block, node->type);

        copy->id = node->id;
        copy->decl = node->decl;
        copy->file = node->file;
        return copy;
    }
```

11.5 IR Fixup

⟨*ir fixup*⟩≡

```
    ⟨blocks_fixup⟩
    ⟨nodes_fixup⟩
    ⟨var_fixup⟩
    ⟨fixup macros⟩
    ⟨fixup⟩
```

Copied objects in `dest` contain pointer fields which, if copied naïvely, would refer to the originals in `src`. The `fixup` function avoids this inconsistency by mapping each through the appropriate array kept by the context.

An array of `exits` is passed when `src` is the callee being inlined into `dest`. The arguments of the former become locals in the latter. Otherwise `src` is one of the callee's lexical descendents, and the variables in its `.closure` require fixup.

⟨*fixup*⟩≡
```
static void fixup(copy_ctx_t *ctx, cfunction_t *dest,
                  cnode_array_t *exits)
{
    cfunction_t *src = ctx->src;
    cblock_t *sblock;
    cvar_t *var;

    if(exits)
    {
        list_foreach_entry(&src->cvar_head, var, cvar_list)
            if(var->local.is_arg)
                ctx->vars[var->id]->local.is_arg = false;
    }
    else
    {
        array_foreach_entry(&src->closure, var)
            array_push(&dest->closure, var_fixup(ctx, var));
    }
    list_foreach_entry(&src->cblock_head, sblock, cblock_list)
    {
        ⟨block fixup⟩
    }
}
```

Each `sblock` in the `src` is paired, via its `.id` and the `.blocks` array, with a corresponding `dblock` in the `dest`. Each `snode` within it is likewise paired with a corresponding `dnode`.

⟨*block fixup*⟩≡
```
cblock_t *dblock = ctx->blocks[sblock->id];
cnode_t *snode;

BLOCKS(pred);
BLOCKS(succ);
list_foreach_entry(&sblock->cnode_head, snode, cnode_list)
{
    cnode_t *dnode = ctx->nodes[snode->id];

    NODES(users);
    switch(snode->type)
    {
        ⟨fixup switch⟩
    }
}
```

This established, a number of macros then assist in the mapping process. The `f` argument names a field. Its value in the destination object will be the copied object which corresponds to the original object that is the field's referent in the source.

`NODE` and `FUNC` each map a pointer to a node or a function, respectively.

⟨*fixup macros*⟩≡
```
#define NODE(f) dnode->f = ctx->nodes[snode->f->id]
#define FUNC(f) dnode->f = ctx->functions[snode->f->id]
```

Arrays of blocks and nodes are mapped elementwise by helper functions, which are called by `BLOCKS` and `NODES`.

⟨*fixup macros*⟩+≡
```
#define BLOCKS(f) dblock->f = blocks_fixup(ctx, &sblock->f)
#define NODES(f) dnode->f = nodes_fixup(ctx, &snode->f)
```

A reference to a variable is also mapped by a helper, invoked by `VAR`.

⟨*fixup macros*⟩+≡
```
#define VAR(f) dnode->f = var_fixup(ctx, snode->f)
```

The `COPY` macro is for fields of any other type; these don't need fixing up.

⟨*fixup macros*⟩+≡
```
#define COPY(f) dnode->f = snode->f
```

`CALL` nodes have `.names`; `CALL_FAST` have `.argbits`. They share `.target` and `.args`.

⟨*fixup switch*⟩≡
```
case CN_CALL:
case CN_CALL_FAST:
    NODE(call.target);
    NODES(call.args);
    if(snode->type == CN_CALL)
        array_copy(&dnode->call.names, &snode->call.names);
    else
        COPY(call.argbits);
    break;
```

`SET` and `BIND` nodes share fields, but the latter may have both or either set to `NULL`.

⟨*fixup switch*⟩+≡
```
case CN_SET:
case CN_BIND:
    if(snode->set.var)
        VAR(set.var);
    if(snode->set.value)
        NODE(set.value);
    break;
```

The `.function`'s backpointer to the `LAMBDA` is set after it's fixed up.

⟨*fixup switch*⟩+≡
```
case CN_LAMBDA:
    FUNC(lambda.function);
    NODES(lambda.closure);
    dnode->lambda.function->node = dnode;
    break;
```

If `exits` was provided, each copied `RETURN` node is recorded within.

⟨*fixup switch*⟩+≡
```
case CN_RETURN:
    NODE(ret.value);
    if(exits)
        array_push(exits, dnode);
    break;
```

No other `.types` of node need particularly special treatment.

⟨*fixup switch*⟩+≡
```
case CN_REF:
    VAR(ref.var);
    break;
case CN_IF:
    NODE(ifelse.cond);
    break;
case CN_PHI:
    NODES(phi.args);
    break;
case CN_BUILTIN:
    COPY(builtin.bi);
    COPY(builtin.optype);
    NODES(builtin.args);
    break;
case CN_COPY:
    NODE(copy.value);
    break;
case CN_CONST:
    COPY(constant);
    break;
```

`blocks_fixup` returns a copy of the `psrc` array, with each element mapped to its corresponding copy in `.blocks`.

⟨*blocks_fixup*⟩≡
```
static cblock_array_t blocks_fixup(copy_ctx_t *ctx, cblock_array_t *psrc)
{
    cblock_array_t dest = ARRAY_INIT;
    cblock_t **ptr;

    array_copy(&dest, psrc);
    array_foreach_ptr(&dest, ptr)
        *ptr = ctx->blocks[(*ptr)->id];
    return dest;
}
```

`nodes_fixup` does the same with `.nodes`, aside from any `NULL`s.

⟨*nodes_fixup*⟩≡
```
static cnode_array_t nodes_fixup(copy_ctx_t *ctx, cnode_array_t *psrc)
{
    cnode_array_t dest = ARRAY_INIT;
    cnode_t **ptr;

    array_copy(&dest, psrc);
    array_foreach_ptr(&dest, ptr)
    {
        if(*ptr)
            *ptr = ctx->nodes[(*ptr)->id];
    }
    return dest;
}
```

Variables are a little more tricky, because they may belong to some ancestor of the function being copied.

This is why `copy_into` calls `copy_vars` in preorder – `var_fixup` can follow the recursion back via the context's `.parent` field, to find the function which binds the variable `var`, and so the `.vars` array that maps it to its copy. If `var` is a global, or not bound by a function we're copying, it doesn't need fixing up.

⟨*var_fixup*⟩≡
```
static cvar_t *var_fixup(copy_ctx_t *ctx, cvar_t *var)
{
    if(cvar_is_global(var))
        return var;
    for(; ctx && var->local.binder != ctx->src; ctx = ctx->parent);
    if(!ctx)
        return var;
    return ctx->vars[var->id];
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "opt.h"
```

Chapter 12

Cell Introduction

A closure captures variables, but a `LAMBDA` node copies values (Section 7.6). A captured variable which is constant only has one value, so these are equivalent. For each of the others, to account for the discrepancy – and allow sharing between closures – the compiler introduces a *cell*, a container for a single value (Section 19.4). The `LAMBDA` then copies a reference to the cell, and the code generated for the function accesses its value indirectly.

⟨*opt_closure.c*⟩≡
  ⟨*includes*⟩
  ⟨*cell builtins*⟩
  ⟨*cell helpers*⟩
  ⟨*cell rewrites*⟩
  ⟨*cfunc_cell_rewrite*⟩
  ⟨*ir_cell_rewrite*⟩
  ⟨*ir_cell_finalize*⟩
  ⟨*ir_cell_intro*⟩

Cell introduction corresponds to the "assignment conversion" of Adams et al. (1986) or the "assignment elimination" of Clinger and Hansen (1994). It is is performed in two phases by `ir_cell_intro`. A closed mutable variable is "celled"; for these, the `cvar_is_celled` predicate returns `true` (Subsection 5.2.2).

⟨*ir_cell_intro*⟩≡
```
cresult ir_cell_intro(cfunction_t *fn)
{
    return ir_cell_rewrite(fn) | ir_cell_finalize(fn);
}
```

The first phase, `ir_cell_rewrite`, transforms each function `fn` with a call to `cfunc_cell_rewrite`; if this adds any new nodes, they're numbered by another call to `cfunc_rdfo`. Child functions are recursively transformed in preorder.

⟨*ir_cell_rewrite*⟩≡
```
static cresult ir_cell_rewrite(cfunction_t *fn)
{
    cresult res = cfunc_cell_rewrite(fn);
    if(changed(res))
        cfunc_rdfo(fn);
    return res | cfunc_map_children(fn, ir_cell_rewrite);
}
```

In the second phase, `ir_cell_finalize` finishes the treatment of celled variables bound by the function `fn`. Their `.decl` and `.is_const` fields are updated to reflect their changed types and immutability. `cfunc_populate_closure` fills the `NULL` elements left for them in the `.closure` arrays of each child function's `LAMBDA` node. They're now eligible for SSA conversion (Chapter 9), which is carried out at this point (when necessary,) before continuing the recursive traversal.

⟨*ir_cell_finalize*⟩≡

```
cresult cfunc_populate_closure(cfunction_t *fn);
static cresult ir_cell_finalize(cfunction_t *fn)
{
    cvar_t *var;
    cresult res;

    list_foreach_entry(&fn->cvar_head, var, cvar_list)
    {
        if(cvar_is_celled(var))
        {
            var->decl = cell_decl_type(var);
            var->is_const = true;
        }
    }
    res = cfunc_map_children(fn, cfunc_populate_closure);
    if(changed(res))
        cfunc_ssa_convert(fn);
    return res | cfunc_map_children(fn, ir_cell_finalize);
}
```

Within the body of a function `fn`, nodes involving celled variables are rewritten by `cfunc_cell_rewrite`. The node list traversal is `_safe`, as we may modify `.cnode_list.next`.

⟨*cfunc_cell_rewrite*⟩≡

```
static cresult cfunc_cell_rewrite(cfunction_t *fn)
{
    cresult res = SUCCESS;
    cblock_t *block;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        cnode_t *node, *tmp;

        list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
        {
            switch(node->type)
            {
                ⟨rewrite switch⟩
            default:
                break;
            }
        }
    }
    return res;
}
```

A `REF` of a celled variable is rewritten by `cell_intro_get`.

⟨*rewrite switch*⟩≡
```
case CN_REF:
{
    cvar_t *var = node->ref.var;
    if(!cvar_is_celled(var))
        continue;
    cell_intro_get(node, var);
    res |= CHANGED;
    break;
}
```

A `BIND` of a celled variable is rewritten by `cell_intro_arg`, if it's introducing an argument, or `cell_intro_local` if not. A `SET` is rewritten by `cell_intro_set`.

⟨*rewrite switch*⟩+≡
```
case CN_SET:
case CN_BIND:
{
    cvar_t *var = node->set.var;
    if(!var || !cvar_is_celled(var))
        continue;
    if(!node->set.value)
        cell_intro_arg(node, var);
    else if(node->type == CN_BIND)
        cell_intro_local(node, var, node->set.value);
    else
        cell_intro_set(node, var, node->set.value);
    res |= CHANGED;
    break;
}
```

12.1 Rewrite Rules

The variable v of type t will hold a value of type `cell(`t`)` instead. Each `REF` of such a variable is followed by an invocation of the `cell_get` `BUILTIN` to extract the value from the cell.

i.	`REF` v	: t
ii.	`REF` v	: `cell(`t`)`
i.	`CELL_GET` ii	: t

⟨*cell rewrites*⟩≡
```
static void cell_intro_get(cnode_t *node, cvar_t *var)
{
    cnode_t *ref = cnode_insert_before(node, CN_REF);

    init_ref(ref, var);
    cnode_reset(node);
    init_cellget(node, var, ref);
}
```

A SET of v first REFs the variable to get the cell before storing the value *val* into it with the **cell_set** BUILTIN.

i.	SET v *val*	
ii.	REF v	: cell(v.decl)
i.	CELL_SET ii *val*	

⟨*cell rewrites*⟩+≡
```
static void cell_intro_set(cnode_t *node, cvar_t *var, cnode_t *val)
{
    cnode_t *ref = cnode_insert_before(node, CN_REF);

    init_ref(ref, var);
    cnode_reset(node);
    init_cellset(node, var, ref, val);
}
```

To rewrite the initial BIND node for a celled argument v, a cell is created by the **cell_make** BUILTIN, initialised by **cell_set** with the value of the argument, then the variable is SET to the cell itself.

i.	BIND v NULL	: t
ii.	BIND v NULL	: t
i.	CELL_MAKE	: cell(t)
iii.	CELL_SET i ii	
iv.	SET v i	

⟨*cell rewrites*⟩+≡
```
static void cell_intro_arg(cnode_t *node, cvar_t *var)
{
    cnode_t *bind = cnode_insert_before(node, CN_BIND);
    cnode_t *cset = cnode_insert_after(node, CN_BUILTIN);
    cnode_t *set = cnode_insert_after(cset, CN_SET);

    assert(var->decl == node->decl);
    init_bind(bind, var, NULL, var->decl);
    init_cellset(cset, var, node, bind);
    init_set(set, var, node);
    cnode_reset(node);
    init_cellmake(node, var);
}
```

The BIND node for a celled local variable v is rewritten in a similar manner, modulo the availability of the initial value *val*.

i.	BIND v *val*	: t
ii.	CELL_MAKE	: cell(t)
iii.	CELL_SET ii *val*	
i.	BIND v ii	: cell(t)

⟨*cell rewrites*⟩+≡
```
static void cell_intro_local(cnode_t *node, cvar_t *var, cnode_t *val)
{
    cnode_t *cmake = cnode_insert_before(node, CN_BUILTIN);
    cnode_t *cset = cnode_insert_before(node, CN_BUILTIN);

    init_cellmake(cmake, var);
    init_cellset(cset, var, cmake, val);
    cnode_reset(node);
    init_bind(node, var, cmake, cmake->decl);
}
```

12.2 Helper Functions

Some helper functions simplify the rewriting process.

⟨*cell helpers*⟩≡
  ⟨*cell_decl_type*⟩
  ⟨*init_ref*⟩
  ⟨*init_set*⟩
  ⟨*init_bind*⟩
  ⟨*init_cellaccess*⟩
  ⟨*init_cellset*⟩
  ⟨*init_cellget*⟩
  ⟨*init_cellmake*⟩

The type of cell required for the variable `var` depends on its declared type, and is returned by `cell_decl_type`.

⟨*cell_decl_type*⟩≡
```
static inline rtype_t *cell_decl_type(cvar_t *var)
    { return rcell_type_create(decl_type(var->decl)); }
```

A rewritten `REF` returns a value of this type.

⟨*init_ref*⟩≡
```
static inline void init_ref(cnode_t *node, cvar_t *var)
{
    node->ref.var = var;
    node->decl = cell_decl_type(var);
}
```

A rewritten `SET` uses its value `val`.

⟨*init_set*⟩≡
```
static inline void init_set(cnode_t *node, cvar_t *var, cnode_t *val)
{
    node->set.var = var;
    if(val)
    {
        node->set.value = val;
        cnode_add_user(node, val);
    }
}
```

A rewritten `BIND` also yields a value of the type `decl`.

⟨*init_bind*⟩≡
```
static inline
void init_bind(cnode_t *node, cvar_t *var, cnode_t *val, rtype_t *decl)
{
    node->type = CN_BIND;
    init_set(node, var, val);
    node->decl = decl;
}
```

A `node` which accesses the celled variable `var` is initialised with `init_cellaccess`. The meaning of `arg1` and `arg2` will depend on the particular `BUILTIN bi`. The `.builtin.optype` field holds the type of the cell.

⟨*init_cellaccess*⟩≡
```
static inline
void init_cellaccess(cnode_t *node, cvar_t *var, const cbuiltin_t *bi,
                     cnode_t *arg1, cnode_t *arg2)
{
    cnode_array_t args = ARRAY_INIT;

    if(arg1)
    {
        array_push(&args, arg1);
        cnode_add_user(node, arg1);
    }
    if(arg2)
    {
        array_push(&args, arg2);
        cnode_add_user(node, arg2);
    }
    node->type = CN_BUILTIN;
    node->builtin.bi = bi;
    node->builtin.args = args;
    node->builtin.optype = cell_decl_type(var);
}
```

An invocation of `cell_make` yields the cell created for `var`.

⟨*init_cellmake*⟩≡
```
static inline void init_cellmake(cnode_t *node, cvar_t *var)
{
    init_cellaccess(node, var, &cell_make, NULL, NULL);
    node->decl = node->builtin.optype;
}
```

`cell_get` yields the value contained in the `cell`.

⟨*init_cellget*⟩≡
```
static inline void init_cellget(cnode_t *node, cvar_t *var, cnode_t *cell)
{
    init_cellaccess(node, var, &cell_get, cell, NULL);
    node->decl = node->builtin.optype->elt;
}
```

`cell_set` stores `val` into `cell`, and doesn't yield a value.

⟨*init_cellset*⟩≡
```
static inline
void init_cellset(cnode_t *node, cvar_t *var, cnode_t *cell, cnode_t *val)
{
    init_cellaccess(node, var, &cell_set, cell, val);
}
```

12.3 Builtins

The `.generate_fn` callbacks for `cell_make`, `cell_get` and `cell_set` are invoked during code generation (Section 15.2), and use the `asm` helper functions (Section 15.3) to emit instructions appropriate to the types of values they manipulate.

`.trans_fn` callbacks are unnecessary, as `ir_optimise` is never called after `ir_cell_intro`. The `.enforce_fn` callback for `cell_set`, called during postpass, ensures that its first argument – the updated value – is of the cell's element type.

⟨*enforce_cell_set*⟩≡
```
static cresult enforce_cell_set(cnode_t *node)
{
    cnode_array_t *args = &node->builtin.args;
    assert(cnode_compat(aref(args, 0), node->builtin.optype));
    return enforce_decl(node, node->builtin.optype->elt, aptr(args, 1), true);
}
```

The `cellmake` instruction (Section 26.4.6) creates a new cell of type `.optype`.

⟨*gen_cell_make*⟩≡
```
static void gen_cell_make(cnode_t *node)
{
    asm_op(OP_cellmake);
    asm_stack(loc_for(node));
    asm_offset(const_index(node->builtin.optype));
}
```

Opcodes for the `cellget` and `cellset` instructions (Subsection 26.6.2) depend on the contained value's size.

⟨*gen_cell_get*⟩≡
```
static void gen_cell_get(cnode_t *node)
{
    asm_op_width(OP_cellget, node->builtin.optype->elt);
    asm_stack(loc_for(node));
    asm_args(&node->builtin.args);
}
```

⟨*gen_cell_set*⟩≡
```
static void gen_cell_set(cnode_t *node)
{
    asm_op_width(OP_cellset, node->builtin.optype->elt);
    asm_args(&node->builtin.args);
}
```

The `cell_set` builtin does not return a value, so has its `.is_void` flag set; no stack location need be allocated for it.

⟨*cell builtins*⟩≡
```
⟨gen_cell_make⟩
static const cbuiltin_t cell_make = {
    &(builtin_ops_t) { .generate_fn = gen_cell_make },
    NULL, "cell_make"
};
⟨gen_cell_get⟩
static const cbuiltin_t cell_get = {
    &(builtin_ops_t) { .generate_fn = gen_cell_get },
    NULL, "cell_get"
};
⟨enforce_cell_set⟩
⟨gen_cell_set⟩
static const cbuiltin_t cell_set = {
    &(builtin_ops_t) { .enforce_fn = enforce_cell_set,
                       .generate_fn = gen_cell_set,
                       .is_void = true },
    NULL, "cell_set"
};
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "opt.h"
#include "vm/vm_ops.h"
#include "gen_code.h"
```

Chapter 13

Postpass

Previous passes (Section 10.3) have recovered the types of values yielded and used by the nodes in the program (Subsection 5.1.4). Code generation imposes additional constraints on certain of these, and `BUILTIN`s can specify their own via callbacks.

The *postpass* performed by this module satisfies these constraints, transforms `CALL`s to `CALL_FAST`, and removes redundant `COPY` nodes.

⟨*opt_post.c*⟩≡
```
⟨post includes⟩
⟨type enforcement⟩
⟨copy optimisation⟩
⟨cfunc_postpass⟩
⟨ir_postpass⟩
```

13.1 Entry Point

The module entry point realises a simple preorder recursive traversal of the functions in the input.

⟨*ir_postpass*⟩≡
```
cresult ir_postpass(cfunction_t *fn)
{
    return cfunc_postpass(fn) | cfunc_map_children(fn, ir_postpass);
}
```

The `cfunc_postpass` function is invoked for each. This runs `cfunc_post_enforce`, then iterates `cfunc_post_copy` to convergence; cleaning up and renumbering if anything `changed`.

⟨*cfunc_postpass*⟩≡
```
static cresult cfunc_postpass(cfunction_t *fn)
{
    cresult r, res = cfunc_post_enforce(fn);

    do res |= (r = cfunc_post_copy(fn));
    while(r == CHANGED);
    if(changed(res))
    {
        cfunc_cleanup(fn);
        cfunc_rdfo(fn);
    }
    return res;
}
```

13.2 Type Enforcement

⟨*type enforcement*⟩≡
  ⟨*enforce_signature*⟩
  ⟨*enforce_fastcall*⟩
  ⟨*enforce_call*⟩
  ⟨*enforce_phi*⟩
  ⟨*cfunc_post_enforce*⟩

`cfunc_post_enforce` is responsible for inserting the type checks and conversions that code generation requires, and visits each `node` in the function `fn`.

⟨*cfunc_post_enforce*⟩≡

```
static cresult cfunc_post_enforce(cfunction_t *fn)
{
    cblock_t *block;
    cnode_t *node, *tmp;
    cresult res = SUCCESS;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
        {
            switch(node->type)
            {
                ⟨post switch⟩
            default:
                break;
            }
        }
    }
    return res;
}
```

The condition of an `IF` condition must have `boolean` type.

⟨*post switch*⟩≡

```
case CN_IF:
    res |= enforce_decl(node, r_type_boolean,
                        &node->ifelse.cond, true);
    break;
```

The value `RETURN`ed from the function must match the declared type of the latter.

⟨*post switch*⟩+≡

```
case CN_RETURN:
    res |= enforce_decl(node, fn->cl_type->sig->ret_type,
                        &node->ret.value, true);
    break;
```

A `CALL` node, if its arguments can be statically matched, may be transformed to become `_FAST`. In any case, its arguments and result value may require conversion according to the calling convention.

⟨*post switch*⟩+≡

```
case CN_CALL:
    res |= enforce_call(node, node->call.target->decl);
    break;
```

A CALL_FAST node is unusual to see here, but can result from a failed inlining attempt (Section 11.1).

⟨*post switch*⟩+≡
```
case CN_CALL_FAST:
    res |= enforce_fastcall(node, node->call.target->decl);
    break;
```

Most SET nodes have been removed; one that remains refers to a global variable. If it has a declared type, this was enforced during prepass (Section 7.4). The remainder have conversions inserted here (boxing scalars, most likely).

⟨*post switch*⟩+≡
```
case CN_SET:
    res |= enforce_decl(node, node->set.var->decl, &node->set.value, true);
    break;
```

Some BUILTINs have specific requirements with respect to the types of their arguments. They can insert any necessary conversions via the optional .enforce_fn callback.

⟨*post switch*⟩+≡
```
case CN_BUILTIN:
{
    const cbuiltin_t *bi = node->builtin.bi;

    if(bi->ops->enforce_fn)
        res |= bi->ops->enforce_fn(node);
    break;
}
```

Codegen expects that the arguments of a PHI all have types compatible with the value yielded by the node itself (Chapter 18).

⟨*post switch*⟩+≡
```
case CN_PHI:
    res |= enforce_phi(node, node->decl);
    break;
```

enforce_phi checks each argument. If a conversion is required, the COPY is inserted at the end of the corresponding predecessor block, to preserve the semantics of the ϕ-function.

⟨*enforce_phi*⟩≡
```
static cresult enforce_phi(cnode_t *node, rtype_t *decl)
{
    int i;
    cresult res = SUCCESS;

    array_foreach(&node->phi.args, i)
    {
        cnode_t **pval = aptr(&node->phi.args, i);

        if(cnode_compat(*pval, decl) == MAYBE)
        {
            cblock_t *pred = aref(&node->block->pred, i);

            *pval = guard_decl(cnode_append(pred, CN_COPY),
                               node, decl, *pval);
            res |= CHANGED;
        }
    }
    return res;
}
```

`enforce_call` first ensures that the target of a `CALL` node is `callable`, should its `type` be known. In this case, enough information is available to attempt argument matching with `call_normalise`; if this succeeds, it becomes a `CALL_FAST` node.

Whatever the outcome, any type conversions required to sastisfy the calling convention will be added by `enforce_signature`.

⟨*enforce_call*⟩≡
```
static cresult enforce_call(cnode_t *node, rtype_t *type)
{
    cresult res = SUCCESS;

    if(type && type != r_type_object && type != r_type_callable)
    {
        if(!rtype_is_callable(type))
        {
            c_error("value of type ‘%s‘ is not callable.", decl_name(type));
            return FAILED;
        }
        res |= call_normalise(node, type->sig);
    }
    return res | enforce_signature(node, type);
}
```

A `CALL_FAST` node must actually supply the mandatory formal arguments of its target, in addition to satisfying the calling convention.

⟨*enforce_fastcall*⟩≡
```
static cresult enforce_fastcall(cnode_t *node, rtype_t *type)
{
    assert(type && type->sig);
    if((node->call.argbits & type->sig->reqbits) != type->sig->reqbits)
    {
        c_error("required argument not supplied.");
        return FAILED;
    }
    return enforce_signature(node, type);
}
```

`enforce_signature` ensures that caller and callee are in agreement about the types of argument and result values – that the actual arguments `args` have types compatible with the formal arguments specified in the callee's signature `sig` (Section 21.2), and that the expected return type `.decl` is compatible with the specified return type (Subsection 5.2.7).

If the `node` is a `CALL`, `sig` is `NULL` and the universal calling convention applies (Subsection 22.2.2). Arguments and result must be reference objects; scalars need to be boxed or unboxed, respectively.

Otherwise it's a `CALL_FAST`; the `type` of the callee specifies the types of argument it expects and the type of value it returns (Subsection 22.2.1).

Actual arguments may be omitted in either case, hence the check for `*ptr`.

⟨*enforce_signature*⟩≡
```
static cresult enforce_signature(cnode_t *node, rtype_t *type)
{
    funsig_t *sig = node->type == CN_CALL ? NULL : type->sig;
    cnode_array_t *args = &node->call.args;
    cresult res = SUCCESS;
    int i;

    assert(node->type == CN_CALL || rtype_is_callable(type));
```

```
    array_foreach(args, i)
    {
        cnode_t **ptr = aptr(args, i);
        rtype_t *decl = sig ? sig->args[i].type : r_type_object;

        if(*ptr)
            res |= enforce_decl(node, decl, ptr, true);
    }
    res |= enforce_val(node, sig ? sig->ret_type : r_type_object);
    return res;
}
```

13.3 Copy Optimisations

`COPY` nodes are generally inserted conservatively, and some occurences may be profitably replaced with equivalent but better performing alternatives.

⟨*copy optimisation*⟩≡
  ⟨*copy_of_const*⟩
  ⟨*update_copy*⟩
  ⟨*cfunc_post_copy*⟩

The `COPY` nodes in function `fn` are tested for redundancy by `cfunc_post_copy`. Since they may be removed, the node list traversal is `safe`.

⟨*cfunc_post_copy*⟩≡

```
static cresult cfunc_post_copy(cfunction_t *fn)
{
    cblock_t *block;
    cnode_t *node, *tmp;
    cresult res = SUCCESS;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
        {
            if(node->type == CN_COPY)
                res |= update_copy(node, node->copy.value);
        }
    }
    return res;
}
```

`update_copy` determines whether the `node` converting a given value `val` is statically unnecessary.

First, the value is checked for compatibility against the type to which it's converted. A `NO` is unlikely and erroneous; a `YES` means that the conversion is redundant and the `COPY` may be removed after being replaced with `val` in its users.

A `MAYBE` confirms the necessity of its presence, but there is a further optimisation to attempt, if it's a `CONST` node that's being copied.

⟨*update_copy*⟩≡

```
static cresult update_copy(cnode_t *node, cnode_t *val)
{
    switch(cnode_compat(val, node->decl))
    {
    case NO:
        c_error("value of type ‘%s‘ found where type ‘%s‘ expected.",
                decl_name(val->decl), decl_name(node->decl));
        return FAILED;
    case YES:
        cnode_replace_in_users(node, val);
        cnode_remove(node);
        return CHANGED;
    case MAYBE:
        break;
    }
    if(val->type == CN_CONST)
        return copy_of_const(node, val->constant);
    return SUCCESS;
}
```

`copy_of_const` tries to perform the conversion specified by the `COPY` `node` ahead of time. The object `obj` is converted to the type `.decl`; if successful, the `COPY` is be replaced with the equivalent `CONST`, via `node_become_constant`.

⟨*copy_of_const*⟩≡

```
static cresult copy_of_const(cnode_t *node, robject_t *obj)
{
    bool valid;

    obj = constant_convert(obj, node->decl, &valid);
    if(valid)
    {
        node_become_constant(node, obj);
        return CHANGED;
    }
    return SUCCESS;
}
```

13.4 Call Normalisation

A `CALL` to a function of known type (i.e. signature) has the opportunity to use the fast calling convention. This requires matching actual to formal arguments at compile-time, a task performed by `call_sequence` and the matcher functions (Section 22.3).

⟨*opt_call.c*⟩≡
  ⟨*call includes*⟩
  ⟨*ir_call_ctx_t*⟩
  ⟨*set_arg*⟩
  ⟨*append_rest*⟩
  ⟨*fail_rest*⟩
  ⟨*call_become_fast*⟩
  ⟨*call_normalise*⟩

This matching will produce an array of argument values in the correct order for generating a fast call sequence (Subsection 15.5.1), stored in the `.args` field of the `ir_call_ctx` structure. Alongside the `.base` context used by the matchers, `.pres` points to the result to be returned, which the callbacks can use to signal errors.

⟨*ir_call_ctx_t*⟩≡
```
typedef struct
{
    call_ctx_t base;
    cnode_array_t args;
    cresult *pres;
} ir_call_ctx_t;
```

`call_normalise` attempts to convert, from `CALL` to `CALL_FAST`, the `node` calling a function with signature `sig`. After `ctx` is initialised it invokes `call_sequence`, passing the actual `args` and `names` supplied by the `.call`.

On success, `call_become_fast` changes the node's `.type` as well as giving it ownership of the matched `.args` (otherwise, these must be explicitly freed). All arguments required by the function must indeed be supplied by the call; if not, the normalisation has `FAILED`.

⟨*call_normalise*⟩≡
```
cresult call_normalise(cnode_t *node, funsig_t *sig)
{
    cresult res = SUCCESS;
    cnode_t **args = adata(&node->call.args);
    rsymbol_t **names = call_has_names(node)
                      ? adata(&node->call.names) : NULL;
    int nargs = alen(&node->call.args);
    ir_call_ctx_t ctx = {
        .base = {
            .posbits = 0,
            .argbits = 0,
            .sig = sig,
            .append_rest = sig->has_rest
                         ? append_rest : fail_rest,
            .set_arg = set_arg
        },
        .args = ARRAY_INIT,
        .pres = &res
    };

    array_resize(&ctx.args, sig->nargs);
    if(call_sequence(&ctx, (void **)args, names, nargs, no_match_rest,
```

```
                    call_match_kwd, call_match_pos, call_match_omit))
    {
        if(sig->reqbits != (ctx.base.argbits & sig->reqbits))
        {
            res |= FAILED;
            c_error("required argument not supplied.");
        }
        else
        {
            res |= CHANGED;
            call_become_fast(node, sig, &ctx.args, ctx.base.argbits);
        }
    }
    else
        array_fini(&ctx.args);
    return res;
}
```

The rest vector "`...`" has statically unknown contents. `no_match_rest` prevents the normalisation if it's encountered, since we can't do anything sensible with it. An enhancement would involve invoking `call_normalise` in an earlier pass where `cell_t .values` are available, to take advantage of constants discovered by SCCP.

The `set_arg` callback is invoked for each matched argument, giving the formal `arg` corresponding to the actual value `val`. Pointer arithmetic yields the index `i` to be assigned in the fast call argument array.

⟨*set_arg*⟩≡

```
static bool set_arg(void *ptr, argdesc_t *arg, void *val)
{
    ir_call_ctx_t *ctx = ptr;
    int i = arg - ctx->base.sig->args;

    aset(&ctx->args, i, val);
    return true;
}
```

If a rest vector is required, the `append_rest` callback returns `false` and the call won't be normalised. This shortcoming could be enhanced by instead inserting the appropriate `BUILTIN` invocations to create a fresh vector and assign its elements.

⟨*append_rest*⟩≡

```
static bool append_rest(void *ptr, rsymbol_t *name, void *val)
    { return false; }
```

If the function doesn't expect a rest vector, extraneous arguments are erroneous, and this is noted by the `fail_rest` callback.

⟨*fail_rest*⟩≡

```
static bool fail_rest(void *ptr, rsymbol_t *name, void *val)
{
    ir_call_ctx_t *ctx = ptr;

    if(name)
        c_error("unknown argument `%s`.", r_symstr(name));
    else
        c_error("too many arguments.");
    *ctx->pres = FAILED;
    return false;
}
```

A `node` is transformed in-place from `CALL` to `CALL_FAST` by `call_become_fast`. The original `.args` array is redundant (as is `.names`), replaced with the normalised `args` array (and `argbits`). Note that the former and latter must contain the same elements (albeit in different order, and with more or fewer `NULL`s), because their `.users` are not updated.

⟨*call_become_fast*⟩≡
```
static void call_become_fast(cnode_t *node, funsig_t *sig,
                             cnode_array_t *args, argbits_t argbits)
{
    array_fini(&node->call.args);
    array_fini(&node->call.names);
    node->type = CN_CALL_FAST;
    node->call.args = *args;
    node->call.argbits = argbits;
}
```

Miscellanea

⟨*post includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "opt.h"
```

⟨*call includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "opt.h"
```

Chapter 14

Value Numbering

Dead code is eliminated, and constants are folded, by earlier passes (Chapter 10). However, the program may also contain redundant computations – nodes (Subsection 5.1.4) that compute the same value (without having other effects). Given a conservative (statically decidable) account of this "sameness", value numbering techniques facilitate their removal.

```
⟨opt_dvn.c⟩≡
  ⟨includes⟩
  ⟨globals⟩
  ⟨hashing⟩
  ⟨numbering⟩
  ⟨cfunc_dvn⟩
  ⟨ir_dvn⟩
```

The `VN` array maps (by `.id`, as usual) each node in the function to some node that computes the same value (possibly to itself). Equivalence is detected via the hash table `tbl`, which is updated during traversal.

```
⟨globals⟩≡
  static cnode_t **VN;
  hashset_t *tbl;
```

14.1 Entry Point

The entry point `ir_dvn` simply invokes `cfunc_dvn` on each function in the tree rooted at `fn`. This pass can't fail and performs its own cleanup, so doesn't return a `cresult`.

```
⟨ir_dvn⟩≡
  void ir_dvn(cfunction_t *fn)
  {
      if(!opt.opt_dvn)
          return;
      cfunc_dvn(fn);
      cfunc_mapc_children(fn, ir_dvn);
  }
```

The dominator tree for the function `fn` is computed by `cfunc_dom` (Chapter 8), the hash `tbl` and `VN` array are allocated, then a call to `dvn` begins the optimisation at the `.entry` block. It doesn't track whether or not it changed the IR; after deallocating the globals, `cfunc_cleanup` and `cfunc_rdfo` are called unconditionally.

⟨*cfunc_dvn*⟩≡
```
static void cfunc_dvn(cfunction_t *fn)
{
    doms_t *dom = cfunc_dom(fn);

    tbl = hashset_create(cnode_hash, cnode_equal);
    VN = xcalloc(fn->nnodes, sizeof(*VN));
    dvn(dom, fn->entry);
    xfree(VN);
    hashset_free(tbl);
    dom_free(dom, fn->nblocks);
    cfunc_cleanup(fn);
    cfunc_rdfo(fn);
}
```

14.2 Dominator Value Numbering

The *dominator value numbering* algorithm detects redundant computations in a block, replacing each with its earlier occurrence in a dominating block (Briggs et al., 1997, Figure 4). It's not as powerful as the global value numbering of Click (1995), but much simpler.

⟨*numbering*⟩≡
```
⟨lookup_vn⟩
⟨lookup_operands⟩
⟨should_record_node⟩
⟨phi_args_valid⟩
⟨record_node⟩
⟨dvn⟩
```

The `dvn` function is similar in structure to `ssa_convert` (Section 9.2). It's called recursively on each `block` in the function. The `scope` array records the nodes made "available" by this block.

⟨*dvn*⟩≡
```
static void dvn(doms_t *dom, cblock_t *block)
{
    cblock_t *nextblk;
    cnode_t *node, *tmp;
    cnode_array_t scope = ARRAY_INIT;

    ⟨value number block⟩
}
```

Each `node` is examined in order. Unless it's a `PHI` for which `phi_args_valid` returns `false`, the nodes that it uses are looked up in `VN` by `lookup_operands`.

The node's entry in the `VN` array is then initialised. If `should_record_node` returns `true`, `record_node` will test for equivalence with the current contents of `tbl`. Otherwise the node is assumed to compute a unique value, and maps to itself.

⟨*value number block*⟩≡

```
list_foreach_entry_safe(&block->cnode_head, node, tmp, cnode_list)
{
    unsigned id = node->id;

    if(node->type != CN_PHI || phi_args_valid(node))
        lookup_operands(node);
    VN[id] = should_record_node(node)
           ? record_node(node, &scope)
           : node;
}
```

The `PHI` `node`s in all successors of `block` are processed, and the corresponding argument of each is looked up in `VN`.

⟨*value number block*⟩+≡

```
array_foreach_entry(&block->succ, nextblk)
{
    int i = index_of_block(&nextblk->pred, block);

    list_foreach_entry(&nextblk->cnode_head, node, cnode_list)
    {
        if(node->type != CN_PHI)
            break;
        lookup_vn(aptr(&node->phi.args, i), node);
    }
}
```

Traversal recurses to the block's `.children` in the dominator tree.

⟨*value number block*⟩+≡

```
array_foreach_entry(&dom->children[block->id], nextblk)
    dvn(dom, nextblk);
```

Upon leaving the block, the `node`s it added to the hash `tbl` are removed.

⟨*value number block*⟩+≡

```
array_foreach_entry(&scope, node)
    hashset_remove(tbl, node);
array_fini(&scope);
```

If the arguments of the given `PHI` `node` have had their entries in `VN` initialised, the `phi_args_valid` helper returns `true` (i.e. in the absence of back edges).

⟨*phi_args_valid*⟩≡

```
static bool phi_args_valid(cnode_t *node)
{
    cnode_t *arg;

    array_foreach_entry(&node->phi.args, arg)
        if(!VN[arg->id])
            return false;
    return true;
}
```

The `should_record_node` helper returns `true` if it's safe for the `node` to be eliminated, should it be found redundant.

Non-constant global variables may, in general, change between accesses by `REF` nodes (any other references remaining are to closed variables – which are all constant, since we introduced cells for those that weren't in Chapter 12).

Side effects are not tracked, so only a `node` without any, as determined by `cnode_is_pure` (Subsection 5.2.4), may be replaced with an equivalent.

⟨*should_record_node*⟩≡
```
static inline bool should_record_node(cnode_t *node)
{
    if(node->type == CN_REF && cvar_is_global(node->ref.var) &&
       !node->ref.var->is_const)
        return false;
    return cnode_is_pure(node, true);
}
```

The return value of `record_node` is stored as the node's `VN` entry. It attempts to insert the `node` into the hash `tbl`.

If an equivalent `val` is already present there, this one is redundant – it's replaced in its users with the earlier occurrence, and removed from the program.

Otherwise the value this `node` computes is unique along this path in the dominator tree; it's recorded in the `scope` array for later removal.

⟨*record_node*⟩≡
```
static inline cnode_t *record_node(cnode_t *node, cnode_array_t *scope)
{
    cnode_t *val = hashset_insert(tbl, node);

    if(val != node)
    {
        cnode_replace_in_users(node, val);
        cnode_remove(node);
        return val;
    }
    array_push(scope, node);
    return node;
}
```

The `lookup_operands` helper just invokes `lookup_vn` on the operands of the given `node` via `cnode_map_used` (Subsection 5.2.5).

⟨*lookup_operands*⟩≡
```
static inline void lookup_operands(cnode_t *node)
{
    cnode_map_used(node, lookup_vn, node);
}
```

`lookup_vn` looks up the element of `VN` that corresponds to `*ptr`, assigning it back to the same place (while tracking users.)

⟨*lookup_vn*⟩≡
```
static void lookup_vn(cnode_t **ptr, void *data)
{
    assert(*ptr);
    cnode_remove_user(data, *ptr);
    assert(VN[(*ptr)->id]);
    *ptr = VN[(*ptr)->id];
    cnode_add_user(data, *ptr);
}
```

14.3 Hashing & Equality

Nodes `n` and `m` are considered equivalent if `cnode_equal(n,m)` returns `true`. The `cnode_hash` function computes a suitable code for storage in a hash table.

⟨*hashing*⟩≡
  ⟨*hash_ptr*⟩
  ⟨*cnode_hash*⟩
  ⟨*cnode_equal*⟩

Nodes which are equal must have equal hash codes. Pointer values can be hashed if they're unique – if no two unequal pointers can be considered to point to the "same" thing.

This is the case for the salient fields of the `cnode_t` structure. Pointers to non-nodes are unique by construction; pointers to nodes are rendered so as `dvn` proceeds. The node `.type` determines which branch of the union is hashed.

⟨*cnode_hash*⟩≡
```
static uint32_t cnode_hash(const void *ptr)
{
    int i;
    const cnode_t *node = ptr;
    uint32_t hash = hash_code(&node->type, sizeof(cnodetype));

    hash_ptr(&node->decl, &hash);
    switch(node->type)
    {
    case CN_CONST: hash_ptr(&node->constant, &hash); break;
    case CN_IF: hash_ptr(&node->ifelse.cond, &hash); break;
    case CN_RETURN: hash_ptr(&node->ret.value, &hash); break;
    case CN_COPY: hash_ptr(&node->copy.value, &hash); break;
    case CN_REF: hash_ptr(&node->ref.var, &hash); break;
    case CN_LAMBDA: hash_ptr(&node->lambda.function, &hash); break;
    case CN_CALL:
    case CN_CALL_FAST:
        hash_ptr(&node->call.target, &hash);
        array_foreach(&node->call.args, i)
        {
            hash_ptr(aptr(&node->call.args, i), &hash);
            if(node->type == CN_CALL && alen(&node->call.names) > 0)
                hash_ptr(aptr(&node->call.names, i), &hash);
        }
        if(node->type == CN_CALL_FAST)
            hash = hash_code_seed(&node->call.argbits,
                                  sizeof(argbits_t), hash);
        break;
    case CN_BIND:
    case CN_SET:
        hash_ptr(&node->set.var, &hash);
        hash_ptr(&node->set.value, &hash);
        break;
    case CN_PHI:
        array_foreach(&node->phi.args, i)
            hash_ptr(aptr(&node->phi.args, i), &hash);
        break;
    case CN_BUILTIN:
        hash_ptr(&node->builtin.bi, &hash);
        hash_ptr(&node->builtin.optype, &hash);
        array_foreach(&node->builtin.args, i)
```

```
            hash_ptr(aptr(&node->builtin.args, i), &hash);
        break;
    }
    return hash;
}
```

The result of hashing the bits of the pointer `ptr` is mixed with the value stored at `*phash` by the `hash_ptr` helper.

⟨*hash_ptr*⟩≡
```
static inline void hash_ptr(const void *ptr, uint32_t *phash)
    { *phash = hash_code_seed(ptr, sizeof(void *), *phash); }
```

`cnode_equal` is the equality predicate for nodes. Again, the node `.type` determines the branch of the union whose fields are tested for equality.

⟨*cnode_equal*⟩≡
```
static bool cnode_equal(const void *x, const void *y)
{
    const cnode_t *nx = x, *ny = y;
    int i;

    if(nx == ny)
        return true;
    if(nx->type != ny->type || nx->decl != ny->decl)
        return false;
    switch(nx->type)
    {
    case CN_CONST: return nx->constant == ny->constant;
    case CN_IF: return nx->ifelse.cond == ny->ifelse.cond;
    case CN_RETURN: return nx->ret.value == ny->ret.value;
    case CN_COPY: return nx->copy.value == ny->copy.value;
    case CN_REF: return nx->ref.var == ny->ref.var;
    case CN_LAMBDA: return nx->lambda.function == ny->lambda.function;
    case CN_CALL:
    case CN_CALL_FAST:
        if(nx->call.target != ny->call.target ||
           alen(&nx->call.args) != alen(&ny->call.args))
            return false;
        array_foreach(&nx->call.args, i)
        {
            if(aref(&nx->call.args, i) != aref(&ny->call.args, i))
                return false;
            if(nx->type == CN_CALL && alen(&nx->call.names) > 0
               && aref(&nx->call.names, i) != aref(&ny->call.names, i))
                return false;
        }
        if(nx->type == CN_CALL_FAST
           && nx->call.argbits != ny->call.argbits)
            return false;
        break;
    case CN_BIND:
    case CN_SET:
        return nx->set.var == ny->set.var && nx->set.value == ny->set.value;
    case CN_PHI:
        if(alen(&nx->phi.args) != alen(&nx->phi.args))
            return false;
        array_foreach(&nx->phi.args, i)
            if(aref(&nx->phi.args, i) != aref(&ny->phi.args, i))
                return false;
```

```
        break;
    case CN_BUILTIN:
        if(nx->builtin.bi != ny->builtin.bi
           || nx->builtin.optype != ny->builtin.optype
           || alen(&nx->builtin.args) != alen(&ny->builtin.args))
            return false;
        array_foreach(&nx->builtin.args, i)
            if(aref(&nx->builtin.args, i) != aref(&ny->builtin.args, i))
                return false;
        break;
    }
    return true;
}
```

Miscellanea

⟨*includes*⟩≡

```
#include "global.h"
#include "ir.h"
#include "opt.h"
```

Chapter 15

Code Generation

The compiler's execution target is a *virtual machine* (Chapter 26) that interprets *bytecode* instructions. The *code generation* subsystem assigns storage locations to the values computed in the input program, inserts copies to replace the ϕ-functions of SSA form, and creates a runtime object containing the generated instruction stream.

⟨*gen_code.c*⟩≡
  ⟨*includes*⟩
  ⟨*context*⟩
  ⟨*type_for*⟩
  ⟨*control flow*⟩
  ⟨*nodes*⟩
  ⟨*blocks*⟩
  ⟨*closure_env*⟩
  ⟨*gen_child_function*⟩
  ⟨*gen_function*⟩

15.1 Entry Point

The subsystem entry point `gen_function` mediates between compiler and runtime, creating an `rfunction_t` (Section 21.5) from a `cfunction_t` (Subsection 5.1.1).

The `live` ranges of the nodes in the function `fn`, computed by `cfunc_liveranges` (Chapter 16), are needed as input by `gen_locations` (Chapter 17), which allocates to each node a specific location in the locals area of the VM stack frame to hold the value it computes. `loc_scalsz` and `loc_sz` receive the number of bytes occupied by scalar values, and all local values, respectively.

If the function captures any lexical values (Section 7.6), they're assigned locations in the closure's environment by `closure_env`. Similarly, `env_scalsz` and `env_sz` receive the number of bytes taken by captured scalars and all captured values.

The `consts` array is initialised; it will serve as the function's constant pool. The call to `gen_code` returns a buffer containing bytecode instructions, generated from the function's body.

Passing these results to `rfunc_create` yields the runtime object `rfn`, which is returned after recursing to child functions, storing the generated objects at their reserved indices in the constant pool.

⟨*gen_function*⟩≡
```
rfunction_t *gen_function(cfunction_t *fn)
{
    liveranges_t *live = cfunc_liveranges(fn);
    op_offset_t loc_scalsz = 0, loc_sz = 0;
    op_stack_t *locs = gen_locations(fn, live, &loc_scalsz, &loc_sz);
```

```
    op_offset_t env_scalsz = 0, env_sz = 0;
    op_offset_t *env_ofs = !cfunc_has_closure(fn) ? NULL
                         : closure_env(fn, &env_scalsz, &env_sz);
    robject_array_t consts = ARRAY_INIT;
    op_code_t *code = gen_code(fn, locs, loc_sz, env_ofs, &consts);
    rfunction_t *rfn = rfunc_create(fn->cl_type, code, loc_scalsz, loc_sz,
                                    env_scalsz, env_sz, &consts);
    cfunction_t *child;

    list_foreach_entry(&fn->cfunc_head, child, cfunc_list)
        rfn->consts[child->id] = gen_child_function(child);
    return rfn;
}
```

If the closure of the `child` function doesn't capture any values, `gen_child_function` can create the `rclosure_t` ahead of time. The bytecode of the parent will refer to it with a `const` instruction instead of a `lambda`. Since each `cfunction_t` is one-to-one with its defining `LAMBDA` node, there's no unnecessary duplication.

⟨*gen_child_function*⟩≡

```
rfunction_t *gen_function(cfunction_t *fn);
static robject_t *gen_child_function(cfunction_t *child)
{
    rfunction_t *rfn = gen_function(child);

    if(cfunc_has_closure(child))
        return (robject_t *)rfn;
    return (robject_t *)rcall_closure_create(rfn->cl_type, rfn);
}
```

15.1.1 Closure Environment

Each element of the `ofs` array records the byte offset in the closure's `.env` buffer of the corresponding value from the `LAMBDA` node's `.closure` array.

Values are stored in a closure environment (Section 21.4) segregated by type. The first iteration of the outer loop operates on scalar values; the second, on pointers.

Offsets are assigned to values in this two-phase traversal, `*pscalsz` and `*psz` receive the sizes of the two classes, and `ofs` is returned.

⟨*closure_env*⟩≡

```
static op_offset_t *closure_env(cfunction_t *fn, op_offset_t *pscalsz,
                                op_offset_t *psz)
{
    op_offset_t *ofs = xcalloc(alen(&fn->closure), sizeof(*ofs));
    op_offset_t sz = 0;

    for(int j = 0; j <= 1; j++)
    {
        int i;

        array_foreach(&fn->closure, i)
        {
            cnode_t *val = aref(&fn->node->lambda.closure, i);
            rtype_t *typ = type_for(val);

            if(j == rtype_is_scalar(typ))
                continue;
            ofs[i] = sz;
```

```
            sz += rtype_eltsz(typ);
        }
        if(j == 0)
            *pscalsz = sz;
        else
            *psz = sz;
    }
    return ofs;
}
```

15.2 Context

Code generation state for the current function is kept in a `gen_ctx_t` context structure.

The `.code` buffer contains the bytecode generated so far. Each element of the `.records` array corresponds to a block in the function.

`.locs`, `.loc_sz`, `.env_ofs` and `.consts` retain the corresponding arguments of `gen_code`; `.nfuncs` and `.closure`, the corresponding fields of the `cfunction_t` being processed.

⟨*gen_ctx_t*⟩≡
```
typedef struct
{
    op_code_array_t code;
    gen_block_t *records;
    op_stack_t *locs;
    op_offset_t loc_sz;
    op_offset_t *env_ofs;
    robject_array_t *consts;
    unsigned nfuncs;
    cvar_array_t *closure;
} gen_ctx_t;
```

The context is global to the subsystem (although only one other module needs access.)

⟨*context*⟩≡
```
gen_ctx_t ctx;
```

A `gen_block_t` structure tracks state for a single basic block. An instruction is addressed with a byte offset into the code buffer. When generating bytecode for a block, the address of its first instruction is stored in the `.addr` field and the `.valid` flag is set.

If a previous control-flow instruction required this address, it will have appended the address of its relevant operand to the list of `.fixups`. These are patched when the block's address is determined.

After generating code for a block, the `.copies` required by SSA deconstruction are appended, in list order.

⟨*gen_block_t*⟩≡
```
typedef struct
{
    op_offset_t addr;
    bool valid;
    list_t copies;
    SLIST(gen_fixup_t) fixups;
} gen_block_t;
```

The `gen_fixup_t` structure requests that the bytecode at `.addr` be overwritten with the start address of the owning block.

⟨*gen_fixup_t*⟩≡
```
typedef struct gen_fixup
{
    SLIST(struct gen_fixup) next;
    op_offset_t addr;
} gen_fixup_t;
```

The `gen_copy_t` structure requests that a `mov` instruction be generated to copy a value of given `.type` from the stack location `.src` to `.dest`.

⟨*gen_copy_t*⟩≡
```
typedef struct gen_copy
{
    list_t list;
    op_stack_t dest, src;
    rtype_t *type;
} gen_copy_t;
```

Records of blocks, and stack locations of nodes, are accessed via the `rec_for` and `loc_for` helpers respectively.

⟨*context helpers*⟩≡
```
static inline gen_block_t *rec_for(cblock_t *block)
    { return &ctx.records[block->id]; }
static inline op_stack_t loc_for(cnode_t *node)
    { return ctx.locs[node->id]; }
```

The `type_for` helper returns the statically determined type of the `node`; or `object`, for those which are neither declared or inferred.

⟨*type_for*⟩≡
```
static inline rtype_t *type_for(cnode_t *node)
    { return decl_type(node->decl); }
```

15.3 Assembly

A set of helper functions are used to write and modify bytecode during generation. `asm_extend` grows the bytecode buffer to encompass an additional `sz` bytes, returning a pointer to the start of the appended block.

⟨*asm_extend*⟩≡
```
static inline uint8_t *asm_extend(size_t sz)
{
    size_t len = asm_len();
    return array_extend(&ctx.code, sz) + len;
}
```

`asm_literal` appends the `sz` bytes from memory referenced by `ptr`.

⟨*asm_literal*⟩≡
```
static inline void asm_literal(void *ptr, size_t sz)
{
    uint8_t *dest = asm_extend(sz);
    memcpy(dest, ptr, sz);
}
```

`asm_op` looks up an opcode in the direct-threading jump table (Section 26.3) and appends the address it finds there to the bytecode buffer. `asm_subop` appends an opcode directly.

⟨*asm_op*⟩≡
```
static inline void asm_subop(op_code_t v)
{
    *(asm_extend(sizeof(op_code_t))) = v;
}
static inline void asm_op(op_code_t v)
{
    *(const void **)(asm_extend(sizeof(vm_instr_t))) = vm_instr[v];
}
```

Some opcodes are grouped by operand width (Section 26.8). `asm_op_width` appends the instruction, with base opcode `op`, applicable to values the same size as those of `type`.

⟨*asm_op_width*⟩≡
```
static inline void asm_op_width(op_code_t op, rtype_t *type)
    { asm_op(op + op_width_ofs[width_code(type)]); }
```

The `width_code` helper returns an index representing the `type`'s size.

⟨*width_code*⟩≡
```
static inline int width_code(rtype_t *type)
{
    switch(rtype_eltsz(type))
    {
    case 1: return 0;
    case 4: return 1;
    case 8: return 2;
    default: return -1;
    } /* NOTREACHED */
}
```

Other opcodes are grouped by operand type. `asm_op_type` appends the instruction, with base opcode `op`, applicable to values of the given scalar `type`.

⟨*asm_op_type*⟩≡
```
static inline void asm_op_type(op_code_t op, rtype_t *type)
    { assert(rtype_is_scalar(type)); asm_op(op + op_type_ofs[rscal_code(type)]); }
```

The conversion opcodes are grouped by both source type and destination type. `asm_op_conv` appends the appropriate opcode to convert from a value of `stype` to a value of `dtype`.

⟨*asm_op_conv*⟩≡
```
static inline void asm_op_conv(rtype_t *stype, rtype_t *dtype)
{
    assert(rtype_is_scalar(stype) && rtype_is_scalar(dtype));
    scalar_code scode = rscal_code(stype), dcode = rscal_code(dtype);
    scalar_code rcode = scode * (SC_MAX-1) + dcode - (scode < dcode);
    asm_op(op_conv_ofs + rcode);
}
```

`asm_stack` appends an operand containing a signed stack location. This will typically be interpreted relative to the VM's base pointer, and specify the location allocated to hold the value computed by a node.

⟨*asm_stack*⟩≡
```
static inline void asm_stack(op_stack_t v)
{
    *(op_stack_t *)(asm_extend(sizeof(v))) = v;
}
```

`asm_offset` appends an operand containing an unsigned offset. These are used to index into the constant pool or the closed environment, or to specify a bytecode address within a function.

⟨*asm_offset*⟩≡
```
static inline void asm_offset(op_stack_t v)
{
    *(op_offset_t *)(asm_extend(sizeof(v))) = v;
}
```

`asm_args` appends the stack location of each node `arg` in the `args` array; a convenience for the `.generate_fn` callback of certain builtin functions (Section 10.5).

⟨*asm_args*⟩≡
```
static inline void asm_args(cnode_array_t *args)
{
    cnode_t *arg;
    array_foreach_entry(args, arg)
        asm_stack(loc_for(arg));
}
```

`asm_patch` writes the relative address `addr` to the bytecode buffer at offset `dest`.

⟨*asm_patch*⟩≡
```
static inline void asm_patch(op_offset_t addr, op_offset_t dest)
    { *(op_offset_t *)(ctx.code.ptr + addr) = dest; }
```

`asm_len` returns the buffer's current length.

⟨*asm_len*⟩≡
```
static inline size_t asm_len()
    { return alen(&ctx.code); }
```

15.4 Blocks

⟨*blocks*⟩≡
  ⟨*gen_copies*⟩
  ⟨*gen_fixups*⟩
  ⟨*gen_block*⟩
  ⟨*gen_code*⟩

The `gen_code` function allocates and populates a buffer with the bytecode instructions generated for the body of the function `fn`.

The first `.nfuncs` elements of the `consts` array are reserved for the corresponding child functions, and will be filled later by `gen_function`.

After the context is initialised, the function is taken out of SSA form by `gen_un_ssa`, which realises each `PHI` node as a sequence of copies (Chapter 18). It may require a temporary location – this is outside the local area of the stack frame, at offset `loc_sz` (and isn't tracked by the garbage collector, but is safe nonetheless, since the copies are always `mov`s that don't allocate.)

Prerequisites now established, each `block` is processed by `gen_block`, starting at the function `.entry` and continuing in reverse-depth-first order (as imposed by `cfunc_rdfo`).

Temporaries and inputs are deallocated, the bytecode buffer is shrunk to size and its contents returned. The `consts` array has also been populated with the unique values used by the `CONST` instructions within.

⟨*gen_code*⟩≡
```
static op_code_t *gen_code(cfunction_t *fn, op_stack_t *locs,
                           op_offset_t loc_sz, op_offset_t *env_ofs,
```

```
                              robject_array_t *consts)
{
    cblock_t *block;

    array_resize(consts, fn->nfuncs);
    ctx = (gen_ctx_t) {
        .code = ARRAY_INIT,
        .locs = locs,
        .records = xcalloc(fn->nblocks, sizeof(gen_block_t)),
        .consts = consts,
        .nfuncs = fn->nfuncs,
        .closure = &fn->closure,
        .loc_sz = loc_sz,
        .env_ofs = env_ofs
    };
    gen_un_ssa(fn, loc_sz);
    list_foreach_entry(&fn->cblock_head, block, cblock_list)
        gen_block(block);
    xfree(ctx.records);
    xfree(locs);
    if(env_ofs)
        xfree(env_ofs);

    if(opt.dbg_dump_ir)
        asm_dump(fn, &ctx.code, consts);

    return array_cede(&ctx.code);
}
```

`gen_block` generates bytecode instructions for a single `block`. The start address `addr` is recorded in the block's record `rec`, the latter being flagged `.valid`. Forward references in earlier code can now be patched by `gen_fixups`.

Each `node` in the block is processed by `gen_node` in control-flow order, after which `gen_copies` appends the copies that `gen_un_ssa` requested. If control flow `fall`s through the end of the block, a transfer to the next is generated by `gen_jump`.

⟨*gen_block*⟩≡

```
static void gen_block(cblock_t *block)
{
    op_offset_t addr = asm_len();
    gen_block_t *rec = rec_for(block);
    bool fall = true;
    cnode_t *node;

    rec->addr = addr;
    rec->valid = true;
    gen_fixups(rec);
    list_foreach_entry(&block->cnode_head, node, cnode_list)
    {
        assert(fall);
        fall = gen_node(block, node);
    }
    // no critical edges; copies after IF can't happen
    assert(fall || list_isempty(&rec->copies));
    gen_copies(rec);
    if(fall)
    {
        assert(alen(&block->succ) == 1);
        gen_jump(block, aref(&block->succ, 0));
```

```
    }
}
```

15.4.1 Fixups & Copies

`gen_fixups` resolves forward references to the block with record `rec`, writing the address of the block's first instruction to each `fixup` address. The list of `.fixups` is empty on return.

⟨*gen_fixups*⟩≡

```
static void gen_fixups(gen_block_t *rec)
{
    gen_fixup_t *fixup;

    slist_while_pop(rec->fixups, fixup, next)
    {
        asm_patch(fixup->addr, rec->addr);
        xfree(fixup);
    }
}
```

`gen_copies` appends, in the requested order, the copies realising the `PHI` nodes in the successor of the block with record `rec`. The list of `.copies` is empty on return.

⟨*gen_copies*⟩≡

```
static void gen_copies(gen_block_t *rec)
{
    gen_copy_t *copy;

    list_while_entry(&rec->copies, copy, list)
    {
        list_remove(&copy->list);
        gen_copy(copy->dest, copy->src, copy->type, copy->type);
        xfree(copy);
    }
}
```

15.4.2 Control Flow

⟨*control flow*⟩≡

⟨*is_fallthrough*⟩
⟨*gen_transfer*⟩
⟨*gen_jump*⟩

The `gen_transfer` function appends an instruction with opcode `op` to transfer control flow from the block currently under construction `to` another block in the same function.

When `op` is a conditional branch, it will take an `extra` operand, emitted before the start address of the target block. If the latter is not yet known, a `fixup` for the address operand will be requested.

⟨*gen_transfer*⟩≡
```
static void gen_transfer(op_code_t op, cblock_t *to, op_stack_t *extra)
{
    gen_block_t *rec = rec_for(to);

    asm_op(op);
    if(extra)
        asm_stack(*extra);
    asm_offset(rec->addr);
    if(!rec->valid)
    {
        gen_fixup_t *fixup = xmalloc(sizeof(*fixup));

        *fixup = (gen_fixup_t) {
            .addr = asm_len() - sizeof(op_offset_t),
            .next = NULL
        };
        slist_push(rec->fixups, fixup, next);
    }
}
```

`gen_jump` appends a `jump` instruction to unconditionally transfer control flow `from` the start of one block `to` the end of another, if necessary.

⟨*gen_jump*⟩≡
```
static void gen_jump(cblock_t *from, cblock_t *to)
{
    if(!is_fallthrough(from, to))
        gen_transfer(OP_jump, to, NULL);
}
```

Control flow falls through block `pred` to reach `succ` if the latter immediately succeeds the former in the function's `.cblock_head` list, since code is generated in list order.

⟨*is_fallthrough*⟩≡
```
static inline bool is_fallthrough(cblock_t *pred, cblock_t *succ)
    { return pred->cblock_list.next == &succ->cblock_list; }
```

15.5 Nodes

⟨*nodes*⟩≡
⟨*const_index*⟩
⟨*gen_literal*⟩
⟨*gen_node_const*⟩
⟨*gen_copy*⟩
⟨*gen_node_copy*⟩
⟨*gen_node_builtin*⟩
⟨*gen_node_if*⟩
⟨*gen_node_return*⟩
⟨*call*⟩
⟨*gen_closure*⟩
⟨*gen_node_lambda*⟩
⟨*variables*⟩

⟨*gen_node*⟩

`gen_node` invokes the appropriate function to generate bytecode for each `.type` of `node`, returning `true` when control flow continues to the next instruction.

⟨*gen_node*⟩≡
```
static bool gen_node(cblock_t *block, cnode_t *node)
{
    switch(node->type)
    {
    case CN_IF: return gen_node_if(node, block);
    case CN_RETURN: return gen_node_return(node);
    case CN_CALL: return gen_node_call_universal(node);
    case CN_CALL_FAST: return gen_node_call_fast(node);
    case CN_LAMBDA: return gen_node_lambda(node);
    case CN_CONST: return gen_node_const(node);
    case CN_SET: return gen_node_set(node);
    case CN_REF: return gen_node_ref(node);
    case CN_COPY: return gen_node_copy(node);
    case CN_BUILTIN: return gen_node_builtin(node);
    default: return true;
    } /* NOTREACHED */
}
```

`CONST` nodes generate `literal` instructions for unboxed scalar constants, and `const` instructions for all others. A constant in bytecode is referred to by its index in the function's constant pool.

⟨*gen_node_const*⟩≡
```
static bool gen_node_const(cnode_t *node)
{
    assert(r_subtypep(r_typeof(node->constant), node->decl));
    if(rtype_is_scalar(node->decl))
    {
        assert(r_typeof(node->constant) == node->decl);
        gen_literal(loc_for(node), BOXPTR(node->constant),
                    node->decl);
    }
    else
    {
        asm_op(OP_const);
        asm_stack(loc_for(node));
        asm_offset(const_index(node->constant));
    }
    return true;
}
```

The source operand for a `literal` instruction, of given `type`, is copied by `gen_literal` from memory at `ptr` into the instruction stream.

⟨*gen_literal*⟩≡
```
static void gen_literal(op_stack_t dest, void *ptr, rtype_t *type)
{
    asm_op_width(OP_literal, type);
    asm_stack(dest);
    asm_literal(ptr, rtype_eltsz(type));
}
```

`const_index` returns the index of the value `val` in the `.consts` array (ignoring the reserved indices); adding it if not found. User constants have been interned in the constant table (Section 3.2) and constants used by the compiler are unique by construction, so pointer equality is equivalent to object equality.

⟨*const_index*⟩≡
```
op_offset_t const_index(void *val)
{
    for(int i = ctx.nfuncs; i < alen(ctx.consts); i++)
        if(aref(ctx.consts, i) == val)
            return i;
    array_push(ctx.consts, val);
    return alen(ctx.consts) - 1;
}
```

A `COPY` node may also convert the copied value from one type to another.

⟨*gen_node_copy*⟩≡
```
static bool gen_node_copy(cnode_t *node)
{
    cnode_t *val = node->copy.value;

    gen_copy(loc_for(node), loc_for(val),
             type_for(node), type_for(val));
    return true;
}
```

The `gen_copy` function generates one or more instructions to copy a value of type `stype` from stack location `src` to `dest`, after which it is to be regarded as having type `dtype`.

Equal types result in a `mov` instruction (operating on the requisite number of bytes). Scalar conversion is realised with a `conv` instruction. Converting an `object` to a scalar requires an `unbox` operation; the converse, allocation of a `box`.

A value of one reference type can be considered to have some other reference type when the former is a subtype of the latter. The `check` instruction ensures that this is the case, followed by a `mov` where necessary.

⟨*gen_copy*⟩≡
```
static void gen_copy(op_stack_t dest, op_stack_t src,
                     rtype_t *dtype, rtype_t *stype)
{
    if(dtype == stype)
        asm_op_width(OP_mov, stype);
    else if(rtype_is_scalar(dtype) && rtype_is_scalar(stype))
        asm_op_conv(stype, dtype);
    else if(rtype_is_scalar(dtype))
        asm_op_type(OP_unbox, dtype);
    else if(rtype_is_scalar(stype))
        asm_op_type(OP_box, stype);
    else
    {
        asm_op(OP_check);
        asm_stack(src);
        asm_offset(const_index(dtype));
        if(src == dest)
            return;
        asm_op_width(OP_mov, r_type_object);
    }
    asm_stack(dest);
    asm_stack(src);
}
```

A `BUILTIN` node defers to the `.generate_fn` callback (Section 10.5) specified by the `cbuiltin_t`. This will invoke the `asm` helpers to generate the instruction or instructions that implement its functionality.

⟨*gen_node_builtin*⟩≡
```
static bool gen_node_builtin(cnode_t *node)
{
    const cbuiltin_t *bi = node->builtin.bi;
    assert(bi->ops && bi->ops->generate_fn);
    bi->ops->generate_fn(node);
    return true;
}
```

An `IF` node generates an `if` instruction, which branches to the consequent block when the condition is `true`. When `false`, control flow is transferred to the alternative block.

(An obvious optimisation would be to reverse the sense of the condition – possibly with a distinct `ifnot` instruction – when that allows control to fall through without an extra `jump`.)

⟨*gen_node_if*⟩≡
```
static bool gen_node_if(cnode_t *node, cblock_t *block)
{
    op_stack_t cond = loc_for(node->ifelse.cond);

    gen_transfer(OP_if, aref(&block->succ, 0), &cond);
    gen_jump(block, aref(&block->succ, 1));
    return false;
}
```

A `RETURN` node generates a `ret` instruction.

⟨*gen_node_return*⟩≡
```
static bool gen_node_return(cnode_t *node)
{
    asm_op(OP_ret);
    asm_stack(loc_for(node->ret.value));
    return false;
}
```

The previous two node types handle control flow themselves, so they return `false` via `gen_node` to `gen_block`.

15.5.1 Call

⟨*call*⟩≡
⟨*gen_node_call_fast*⟩
⟨*gen_arg_rest*⟩
⟨*gen_arg_kwd*⟩
⟨*gen_arg_pos*⟩
⟨*gen_arg_omit*⟩
⟨*gen_node_call_universal*⟩

The sequence of instructions generated by a `CALL_FAST` node implements the fast call convention (Subsection 22.2.1). The `target` function's signature `sig` is known, and the actual arguments have already been matched to their formal counterparts (Section 13.4).

The initial `frame` instruction clears a call frame atop the stack of size `.argsz` and copies, as the implicit first argument, the `.argbits` from the call `node`. Each actual argument is `move`d into place at the correct `.offset` (when not omitted.)

This is inefficient when the values aren't live beyond the call – the location allocator could be improved to also operate on outgoing arguments, at the cost of making `frame` explicit in the IR.

A `call_fast` instruction invokes the `target` function; the following `complete` instruction places the value returned at the stack location `dest`.

⟨*gen_node_call_fast*⟩≡
```
static bool gen_node_call_fast(cnode_t *node)
{
    op_stack_t target = loc_for(node->call.target);
    op_stack_t dest = loc_for(node);
    funsig_t *sig = type_for(node->call.target)->sig;
    int i;
    assert(rtype_is_callable(type_for(node->call.target)));
    assert(alen(&node->call.args) == sig->nargs);
    assert((node->call.argbits & sig->reqbits) == sig->reqbits);

    asm_op(OP_frame);
    asm_offset(sig->argsz);
    asm_literal(&node->call.argbits, sizeof(argbits_t));
    array_foreach(&node->call.args, i)
    {
        cnode_t *arg = aref(&node->call.args, i);
        op_stack_t ofs = sig->args[i].offset;

        if(!arg)
            continue;
        gen_copy(ctx.loc_sz + ofs, loc_for(arg),
                 type_for(arg), type_for(arg));
    }
    asm_op(OP_call_fast);
    asm_stack(target);
    asm_op_width(OP_complete, type_for(node));
    asm_stack(dest);
    return true;
}
```

The instructions generated by a `CALL` node enact the universal call convention (Subsection 22.2.2). No properties of the `target` function are assumed; argument matching, type checking and conversion all occur during execution.

A `call_uni` instruction invokes the `target` after forming the argument frame at runtime. Sub-operations to perform this task are emitted by the `gen_arg` callbacks under control of the `call_sequence` function (Section 22.3); a `call_end` sub-op terminates the sequence.

The following `complete_box` instruction places the value returned at the stack location `dest`, boxing it first if the `target`'s signature indicates a scalar return type.

⟨*gen_node_call_universal*⟩≡
```
static bool gen_node_call_universal(cnode_t *node)
{
    op_stack_t target = loc_for(node->call.target);
```

```
    op_stack_t dest = loc_for(node);
    cnode_t **args = adata(&node->call.args);
    int nargs = alen(&node->call.args);
    rsymbol_t **names = call_has_names(node)
                      ? adata(&node->call.names)
                      : NULL;

    asm_op(OP_call_uni);
    asm_stack(target);
    call_sequence(NULL, (void **)args, names, nargs,
                  gen_arg_rest, gen_arg_kwd, gen_arg_pos, gen_arg_omit);
    asm_subop(OP_call_end);
    asm_op(OP_complete_box);
    asm_stack(target);
    asm_stack(dest);
    return true;
}
```

The `call_pos` sub-op specifies a positional argument with actual value given by the node `arg`.

⟨*gen_arg_pos*⟩≡
```
static bool gen_arg_pos(void *ptr, void *arg)
{
    asm_subop(OP_call_pos);
    asm_stack(loc_for(arg));
    return true;
}
```

The`call_kwd` sub-op specifies a named argument, with the `name` a constant symbol and the actual value given by `arg`.

⟨*gen_arg_kwd*⟩≡
```
static bool gen_arg_kwd(void *ptr, rsymbol_t *name, void *arg)
{
    op_offset_t idx = const_index(name);

    asm_subop(OP_call_kwd);
    asm_offset(idx);
    asm_stack(loc_for(arg));
    return true;
}
```

The `call_omit` sub-op marks an omitted positional argument.

⟨*gen_arg_omit*⟩≡
```
static bool gen_arg_omit(void *ptr)
{
    asm_subop(OP_call_omit);
    return true;
}
```

The `call_rest` sub-op passes the value of `arg` as the rest vector "`...`" to the callee.

⟨*gen_arg_rest*⟩≡
```
static bool gen_arg_rest(void *ptr, void *arg)
{
    asm_subop(OP_call_rest);
    asm_stack(loc_for(arg));
    return true;
}
```

15.5.2 Lambda

If the child function `fn` defined by a `LAMBDA` node captures any values, a `lambda` instruction is generated. When executed, this will create a closure from the `rfunction_t` at the constant pool index corresponding to the child's `.id`. It's followed by a sequence of sub-operations, generated by `gen_closure`, that copy the captured values into the closed environment.

Otherwise, a `const` instruction retrieves the closure itself from the constant pool at the corresponding reserved index, where it was stashed by `gen_child_function`.

⟨*gen_node_lambda*⟩≡
```
static bool gen_node_lambda(cnode_t *node)
{
    cfunction_t *fn = node->lambda.function;
    bool clo = cfunc_has_closure(fn);
    int id = fn->id;
    assert(alen(&node->lambda.closure) == alen(&fn->closure));
    asm_op(clo ? OP_lambda : OP_const);
    asm_stack(loc_for(node));
    asm_offset(id);
    if(clo)
        gen_closure(node, fn);
    return true;
}
```

The values captured by the child function `fn` are copied into the closure with `lambda_mov` sub-ops in the same order as their offsets are computed in `closure_env`. This fills the closure in two phases; first with the scalars, then the pointers. The sequence is terminated with a `lambda_end`.

⟨*gen_closure*⟩≡
```
static void gen_closure(cnode_t *node, cfunction_t *fn)
{
    for(int j = 0; j <= 1; j++)
    {
        int i;

        array_foreach(&fn->closure, i)
        {
            cnode_t *val = aref(&node->lambda.closure, i);
            assert(val);
            rtype_t *typ = type_for(val);

            if(j == rtype_is_scalar(typ))
                continue;
            asm_subop(OP_lambda_mov8 + width_code(typ));
            asm_stack(loc_for(val));
        }
    }
    asm_subop(OP_lambda_end);
}
```

15.5.3 Variables

⟨*variables*⟩≡
  ⟨*gen_setvar*⟩
  ⟨*gen_node_set*⟩
  ⟨*gen_getvar*⟩
  ⟨*env_offset*⟩
  ⟨*gen_node_ref*⟩

By now, `SET` nodes can only refer to global variables.

A `GLOBAL_INT` variable is defined by the node `.intl.set` – at this point, a `defvar` instruction will, when executed, add it to the global environment; its name and declared type (if any) are specified by constants.

A `GLOBAL_EXT` variable already exists in the global environment, but may or may not have a declared type.

Either way, `gen_setvar` will generate an appropriate `set` instruction.

⟨*gen_node_set*⟩≡
```
static bool gen_node_set(cnode_t *node)
{
    cvar_t *var = node->set.var;
    assert(r_subtypep(decl_type(node->set.value->decl), decl_type(var->decl)));
    switch(var->type)
    {
    case GLOBAL_INT:
    {
        bool is_def = var->intl.set == node;

        if(is_def)
        {
            asm_op(OP_defvar);
            asm_offset(const_index(var->name));
            asm_offset(const_index(decl_type(var->decl)));
        }
        gen_setvar(var, node->set.value, true, is_def & var->is_const);
        break;
    }
    case GLOBAL_EXT:
    {
        bool has_decl = var->extl.global ? (assert(var->extl.global->decl),true) : false;
        gen_setvar(var, node->set.value, has_decl, false);
        break;
    }
    default: assert(!"reached"); break; /* NOTREACHED */
    }
    return true;
}
```

When `has_decl` is `true`, the variable `var` has a declared type; it is assigned a value with the `setvar` instruction appropriate to its size. An undeclared variable uses the `setvar_uni` instruction instead.

The assigned value is given by the node `val`, the variable `.name` specified by a constant. If `has_decl`, the extra operand `is_def` is `true` when this is the initialising definition of a constant variable. It will set the `.is_const` flag and prevent further updates (Section 19.5).

⟨*gen_setvar*⟩≡
```
static void gen_setvar(cvar_t *var, cnode_t *val,
                       bool has_decl, bool is_def)
{
    if(has_decl)
        asm_op_width(OP_setvar, decl_type(var->decl));
    else
        asm_op(OP_setvar_uni);
    asm_stack(loc_for(val));
    asm_offset(const_index(var->name));
    if(has_decl)
        asm_offset(is_def);
}
```

`REF` nodes may refer to globals or captured lexical variables. `gen_getvar` handles the former case; in the latter, the `env` instruction will extract the value of the variable `var` from the closed environment at the offset given by `env_offset`.

⟨*gen_node_ref*⟩≡
```
static bool gen_node_ref(cnode_t *node)
{
    cvar_t *var = node->ref.var;

    switch(var->type)
    {
    case GLOBAL_INT:
        gen_getvar(var, node, true);
        break;
    case GLOBAL_EXT:
        gen_getvar(var, node, var->extl.global);
        break;
    default:
        asm_op_width(OP_env, type_for(node));
        asm_stack(loc_for(node));
        asm_offset(env_offset(var));
        break;
    }
    return true;
}
```

The elements of the `.env_ofs` array, the function's `LAMBDA` node's `.closure` array, and so the function's `.closure` array, are in the same order. The index of the variable `var` in the last is thus the index in the first at which its offset is stored.

⟨*env_offset*⟩≡
```
static inline op_offset_t env_offset(cvar_t *var)
    { return ctx.env_ofs[index_of_var(ctx.closure, var)]; }
```

Again, `has_decl` is `true` if the global variable `var` was declared; if so, the appropriate `getvar` instruction will be generated. An undeclared global will generate the `getvar_uni` instruction instead. Again, the `.name` is specified by a constant.

⟨*gen_getvar*⟩≡
```
static void gen_getvar(cvar_t *var, cnode_t *node, bool has_decl)
{
    if(has_decl)
        asm_op_width(OP_getvar, type_for(node));
    else
        asm_op(OP_getvar_uni);
    asm_stack(loc_for(node));
    asm_offset(const_index(var->name));
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "opt.h"
#include "vm/vm_ops.h"
#include "gen.h"
#include "gen_range.h"
#include "gen_code.h"
void gen_un_ssa(cfunction_t *fn, op_offset_t temploc); // HACK
```

⟨*gen_code.h*⟩≡
```
⟨gen_copy_t⟩
⟨gen_fixup_t⟩
⟨gen_block_t⟩
⟨gen_ctx_t⟩
⟨externs⟩
⟨context helpers⟩
⟨asm_len⟩
⟨asm_extend⟩
⟨asm_literal⟩
⟨asm_op⟩
⟨asm_stack⟩
⟨asm_offset⟩
⟨width_code⟩
⟨asm_op_width⟩
⟨asm_op_type⟩
⟨asm_op_conv⟩
⟨asm_args⟩
⟨asm_patch⟩
op_offset_t const_index(void *val);
void asm_dump(cfunction_t *fn, op_code_array_t *code, robject_array_t *consts); // DEBUG
```

⟨*externs*⟩≡
```
extern gen_ctx_t ctx;
```

Chapter 16

Live Range Analysis

A value is *live* at every point in the program that lies between its definition and some subsequent use (Brandner et al., 2011). This module computes the *live range* of each node in the input program. The values produced by two nodes can be stored in the same location if their live ranges do not intersect.

Location allocation (Chapter 17) takes advantage of this fact to reduce the size of stack frame required for each function; via processor cache effects, this is expected to improve run-time performance. Dead values of reference type may also be reclaimed more promptly by the garbage collector.

⟨*gen_range.c*⟩≡
  ⟨*includes*⟩
  ⟨*intervals*⟩
  ⟨*ranges*⟩
  ⟨*cfunc_liveranges*⟩

16.1 Points, Intervals & Ranges

Nodes (Subsection 5.1.4) are taken as defining program points. `cfunc_rdfo` (Section 7.5) has assigned to them `.id` numbers in reverse-depth-first order.

The `range_t` structure descrives the live range of a single `.node`. The `.it_head` list contains the non-contiguous intervals, ordered by increasing start point, within which the value produced by the node is live. The start of the first interval is stored in `.start`; the end of the last, in `.end`. When a location is assigned or allocated for the node, it's stored in `.loc`.

⟨*range_t*⟩≡
```
typedef struct
{
    cnode_t *node;
    list_t it_head;
    list_t range_list;
    unsigned start, end;
} range_t;
```

An `interval_t` structure denotes the half-open interval [`.start`, `.end`). It's linked, through its `.it_list` field, to the `.it_head` of its containing range.

⟨*interval_t*⟩≡
```
typedef struct
{
    unsigned start, end;
    list_t it_list;
} interval_t;
```

⟨*intervals*⟩≡
  ⟨*interval_covers*⟩
  ⟨*interval_create_after*⟩
  ⟨*interval_add*⟩
  ⟨*interval_extend*⟩

The `interval_covers` predicate returns `true` if and only if the interval `it` covers the given `point`.

An interval ends at the use of a value, but does not cover it – for example, if `b` is not live beyond the assignment `a = fn(b)`, `a` and `b` can occupy the same location.

⟨*interval_covers*⟩≡
```
static inline bool interval_covers(interval_t *it, unsigned point)
    { return it->start <= point && point < it->end; }
```

`interval_add` returns an interval in `range` that ends at or covers the `point` – finding one which already exists, or creating and adding a new interval of zero length.

⟨*interval_add*⟩≡
```
static interval_t *interval_add(range_t *range, unsigned point)
{
    interval_t *prev;

    list_foreach_entry_reverse(&range->it_head, prev, it_list)
    {
        if(prev->end < point)
            break;
        else if(prev->start <= point)
            return prev;
    }
    return interval_create_after(range, prev, point);
}
```

`interval_create_after` constructs the interval [`point`, `point`), inserting it into `range` after the interval `prev`. This is chosen appropriately by the caller so that the `.it_head` list is maintained in sorted order.

⟨*interval_create_after*⟩≡
```
static interval_t *interval_create_after(range_t *range, interval_t *prev,
                                          unsigned point)
{
    interval_t *it = xmalloc(sizeof(*it));

    it->start = it->end = point;
    list_init(&it->it_list);
    list_add(&prev->it_list, &it->it_list);
    return it;
}
```

`interval_extend` extends the `range`'s interval `it` backwards, to start at or cover `point`. If there is an adjacent interval `prev` in that direction, and it would overlap, the two are coalesced (`it`'s removed in favour of updating `prev`) and the result returned.

⟨*interval_extend*⟩≡
```
static interval_t *interval_extend(range_t *range, interval_t *it,
                                   unsigned point)
{
    if(it->it_list.prev != &range->it_head)
    {
        interval_t *prev = list_entry(it->it_list.prev, it, it_list);

        if(prev->end >= point)
        {
            assert(prev->start <= point);
            prev->end = it->end;
            list_remove(&it->it_list);
            xfree(it);
            return prev;
        }
    }
    if(it->start > point)
        it->start = point;
    return it;
}
```

16.2 Live Ranges

The `liveranges_t` structure holds the information required for location allocation, as well as its results. The elements in the `.ranges` array correspond to the nodes in the function being compiled. They're linked, in order of increasing `.start` point, to the `.range_head` list.

⟨*liveranges_t*⟩≡
```
typedef struct
{
    range_t *ranges;
    list_t range_head;
} liveranges_t;
```

The `live range_t` of a `node` is accessed via the `range_for` helper.

⟨*range_for*⟩≡
```
static inline range_t *range_for(liveranges_t *live, cnode_t *node)
    { return &live->ranges[node->id]; }
```

⟨*ranges*⟩≡
  ⟨*range_covers*⟩
  ⟨*skip_empty*⟩
  ⟨*range_extend*⟩
  ⟨*range_free*⟩
  ⟨*range_extents*⟩
  ⟨*liveranges_compute*⟩
  ⟨*liveranges_free*⟩

A `point` is covered by a `range` if and only it's covered by one of the latter's intervals.

⟨*range_covers*⟩≡
```
bool range_covers(range_t *range, unsigned point)
{
    interval_t *it;

    list_foreach_entry(&range->it_head, it, it_list)
    {
        if(interval_covers(it, point))
            return true;
        if(it->start > point)
            return false;
    }
    return false;
}
```

The module entry point `cfunc_liveranges` constructs, populates and returns the `live` ranges structure for the given function `fn`.

⟨*cfunc_liveranges*⟩≡
```
liveranges_t *cfunc_liveranges(cfunction_t *fn)
{
    liveranges_t *live = xmalloc(sizeof(*live));

    *live = (liveranges_t) {
        .ranges = xcalloc(fn->nnodes, sizeof(range_t))
    };
    list_init(&live->range_head);
    liveranges_compute(fn, live);
    return live;
}
```

When code generation is complete, the structure is deallocated with `liveranges_free`.

⟨*liveranges_free*⟩≡
```
void liveranges_free(cfunction_t *fn, liveranges_t *live)
{
    range_t *range;

    for(int i = 0; i < fn->nnodes; i++)
    {
        range = &live->ranges[i];

        if(!list_isempty(&range->it_head))
            range_free(range);
    }
    xfree(live->ranges);
    xfree(live);
}
```

Freeing a `range` entails freeing its consitutent intervals.

⟨*range_free*⟩≡
```
static void range_free(range_t *range)
{
    interval_t *it, *itmp;

    list_foreach_entry_safe(&range->it_head, it, itmp, it_list)
        xfree(it);
    //list_init(&range->it_head);
    //list_remove(&range->range_list);
}
```

16.2.1 Path Exploration

The `liveranges_compute` function discovers the `live` ranges of the nodes in the function `fn`. They're visited in list order, which is the same order as their `.id` numbers (again, thanks to `cfunc_rdfo`).

⟨*liveranges_compute*⟩≡
```
static void liveranges_compute(cfunction_t *fn, liveranges_t *live)
{
    cblock_t *block;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        cnode_t *node;

        list_foreach_entry(&block->cnode_head, node, cnode_list)
        {
            ⟨compute range⟩
        }
    }
}
```

Each `node` has a corresponding `range`. If the former doesn't yield a value, no further work need be done for the latter.

⟨*compute range*⟩≡
```
range_t *range = range_for(live, node);
cnode_t *user;

list_init(&range->it_head);
list_init(&range->range_list);
if(!cnode_yields_value(node))
    continue;
```

Otherwise, the live range is computed in a backwards traversal, beginning at each `use` of the node and ending at the node itself (Brandner et al., 2011, Section 5, Algorithm 6). This is performed by `range_extend`, which is passed the block from which to `begin`, and an interval `it` which ends at the point of use.

If the node is used as an argument of a `PHI`, it is is live out of the corresponding predecessor block, so traversal `begin`s there. The interval `it` covers the `.end` of the block.

Otherwise, the interval ends at (but does not cover) the `use` itself, and traversal `begin`s at its containing block.

⟨*compute range*⟩+≡
```
range->node = node;
array_foreach_entry(&node->users, user)
{
    cblock_t *begin;
    interval_t *it;

    if(user->type == CN_PHI)
    {
        int i = index_of_node(&user->phi.args, node);
        assert(i >= 0);
        begin = skip_empty(aref(&user->block->pred, i));
        it = interval_add(range, begin->end + 1);
    }
    else
    {
        begin = user->block;
        it = interval_add(range, user->id);
    }
    range_extend(range, begin, node, it);
}
```

Even if the node isn't used, its value still has to go somewhere; a zero-length interval is added to its range in this case.

⟨*compute range*⟩+≡
```
if(!cnode_is_used(node))
    interval_add(range, node->id);
```

The `.start` and `.end` of the range are set by `range_extents`. Since nodes are visited in order of increasing `.id`, their ranges are appended to the `.range_list` in order also.

⟨*compute range*⟩+≡
```
range_extents(range);
list_add_before(&live->range_head, &range->range_list);
```

Empty blocks have been inserted on critical edges (Section 7.3). They hold no program points so can be ignored when encountered, a task performed by the `skip_empty` helper.

⟨*skip_empty*⟩≡
```
static inline cblock_t *skip_empty(cblock_t *block)
{
    if(list_isempty(&block->cnode_head))
        return aref(&block->pred, 0);
    return block;
}
```

The `range_extend` function walks backwards in the control-flow graph towards a `node`, growing its live `range` along the way. The interval `it` ends at or covers the end of the `block` being processed.

This is based on the algorithm described in Brandner et al. (2011, Section 5.2) adapted to operate on live ranges instead of live sets, in a similar vein to Mössenböck and Pfeiffer (2002, Section 4.4).

⟨*range_extend*⟩≡
```
static void range_extend(range_t *range, cblock_t *block, cnode_t *node,
                         interval_t *it)
{
    unsigned point;
    cblock_t *pred;

    ⟨extend range⟩
}
```

If the `node` is defined in this `block`, it provides the target program `point`. Otherwise the traversal will continue upwards, so the target `point` is given by the block `.start` – this is also the case if it's a `PHI` node, as the result of a ϕ-function is live into its containing block (Brandner et al., 2011, Section 4).

⟨*extend range*⟩≡
```
if(node->block == block && node->type != CN_PHI)
    point = node->id;
else
    point = block->start;
```

If continuing up this path won't extend the interval, it's already been covered by another invocation, so this one is done.

⟨*extend range*⟩+≡
```
if(interval_covers(it, point))
    return;
```

The interval is extended backward to cover the target point.

⟨*extend range*⟩+≡
```
interval_extend(range, it, point);
```

If the definition has been reached, the traversal is complete.

⟨*extend range*⟩+≡
```
if(node->block == block)
    return;
```

Otherwise, each `pred`ecessor of the `block` is recursively traversed with an interval covering its `.end`.

⟨*extend range*⟩+≡
```
array_foreach_entry(&block->pred, pred)
{
    pred = skip_empty(pred);
    it = interval_add(range, pred->end + 1);
    range_extend(range, pred, node, it);
}
```

The `range_extents` helper sets the overall extrema of the `range`. `first` and `last` may be the same interval.

⟨*range_extents*⟩≡
```
static void range_extents(range_t *range)
{
    interval_t *first = NULL, *last = NULL;

    assert(!list_isempty(&range->it_head));
    first = list_entry(range->it_head.next, first, it_list);
    last = list_entry(range->it_head.prev, last, it_list);
    range->start = first->start;
    range->end = last->end;
    assert(range->start <= range->end);
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
#include "gen.h"
#include "gen_range.h"
```

⟨*gen_range.h*⟩≡
```
⟨interval_t⟩
⟨range_t⟩
⟨liveranges_t⟩
⟨range_for⟩
bool range_covers(range_t *range, unsigned point);
void liveranges_free(cfunction_t *fn, liveranges_t *live);
liveranges_t *cfunc_liveranges(cfunction_t *fn);
op_stack_t *gen_locations(cfunction_t *fn, liveranges_t *live,
                          op_offset_t *pscalsz, op_offset_t *psz); // HACK
```

Chapter 17

Location Allocation

The values that the virtual machine (Chapter 26) operates upon are stored on a *stack*. A stack location is addressed with a signed offset from the VM's *base pointer* register (Section 26.1). Location allocation is the process of assigning stack locations to the values produced by the nodes in a function, according to their types and lifetimes.

⟨*gen_loc.c*⟩≡
  ⟨*includes*⟩
  ⟨*preassignment*⟩
  ⟨*local_t*⟩
  ⟨*linear scan*⟩
  ⟨*concrete locations*⟩
  ⟨*init_locs*⟩
  ⟨*gen_locations*⟩

17.1 Abstract Locations

Before their concrete stack locations can be computed, each node is first assigned an *abstract location*, represented as a `loc_t` structure. Its `.index` is interpreted relative to the *storage base* specified by the `.base` field. The storage base chosen for a node depends on the node's size, type and usage. Two nodes with the same `.index` and `.base` will occupy the same concrete location.

⟨*loc_t*⟩≡
```
typedef struct
{
    locbase base;
    unsigned index;
} loc_t;
```

BASE_8, _32, _64 and _PTR denote storage for local scalar booleans, integers, doubles, and pointers to objects, respectively. The formal arguments to a function are stored at BASE_ARG. BASE_HINT is a request to the allocator that the value be stored at the same location, if possible, as that of another node. Values without special requirements are provisionally initialised to BASE_NONE.

⟨*locbase*⟩≡
```
typedef enum
{
    BASE_8 = SC_BOOLEAN,
    BASE_32 = SC_INT,
    BASE_64 = SC_DOUBLE,
    BASE_PTR,
    BASE_ARG,
    BASE_HINT,
    BASE_NONE
} locbase;
```

Each node which is UNASSIGNED is allocated a location at one of the LOCAL_BASES. The SCALAR_BASES are segregated from BASE_PTR, to simplify garbage collection.

⟨*locbase*⟩+≡
```
#define LOCAL_BASES BASE_ARG
#define SCALAR_BASES (BASE_64+1)
#define UNASSIGNED BASE_HINT
```

The storage base suitable for the type of value produced by a node is returned by the base_for helper function.

⟨*base_for*⟩≡
```
static inline locbase base_for(cnode_t *node)
{
    rtype_t *type = decl_type(node->decl);

    return rtype_is_scalar(type)
        ? rscal_code(type)
        : BASE_PTR;
}
```

The size_base helper returns the number of bytes taken by a value in a storage base.

⟨*size_base*⟩≡
```
static inline size_t size_base(locbase base)
{
    static const int base_sizes[] = { 1, 4, 8, sizeof(robject_t *) };
    return base_sizes[base];
}
```

17.2 Entry Point

The module entry point, gen_locations, computes the bp-relative offset in the VM stack for each node in the function fn. The live ranges (computed in Chapter 16) provide the lifetime information for each node; the locs array stores their abstract locations.

Some nodes are preassigned specific locations by args_assign and phi_assign. Locations for the remainder are allocated by loc_alloc which also returns, in the nlocs array, the number of values stored at each of the LOCAL_BASES. loc_bases then computes the offsets and sizes of these, writing the former to the base_ofs array and the latter to *pscalsz and *psz.

The concrete stack offset of each node can then be computed by `loc_offsets`; the resulting `ofs` array, indexed in the usual fashion by node `.id`, being returned as the result.

⟨*gen_locations*⟩≡
```
op_stack_t *gen_locations(cfunction_t *fn, liveranges_t *live,
                          op_offset_t *pscalsz, op_offset_t *psz)
{
    loc_t *locs = init_locs(fn);
    op_offset_t base_ofs[LOCAL_BASES];
    unsigned nlocs[LOCAL_BASES];
    op_stack_t *ofs;

    args_assign(fn, locs);
    phis_assign(fn, live, locs);
    loc_alloc(fn, live, locs, nlocs);
    loc_bases(base_ofs, nlocs, pscalsz, psz);
    ofs = loc_offsets(fn, base_ofs, locs);
    liveranges_free(fn, live);
    xfree(locs);
    return ofs;
}
```

Every node begins with location `.base` set to `BASE_NONE`.

⟨*init_locs*⟩≡
```
static inline loc_t *init_locs(cfunction_t *fn)
{
    loc_t *locs = xcalloc(fn->nnodes, sizeof(*locs));
    for(int i = 0; i < fn->nnodes; i++)
        locs[i].base = BASE_NONE;
    return locs;
}
```

17.3 Preassignment

The `BIND` nodes corresponding to the formal arguments of the function are found at known offsets in the incoming call frame. `PHI` nodes and their arguments are linked by a mechanism analogous to the "register hints" of Wimmer and Franz (2010).

⟨*preassignment*⟩≡
  ⟨*args_assign*⟩
  ⟨*cmp_node_id*⟩
  ⟨*phi_assign_hints*⟩
  ⟨*phis_assign*⟩

args_assign assigns to `BIND` nodes, in the order they appear in the `.entry` block of function `fn`, an `.index` at `BASE_ARG` (no `BIND` nodes appear elsewhere; they were removed during SSA conversion).

⟨*args_assign*⟩≡
```
static void args_assign(cfunction_t *fn, loc_t *locs)
{
    int argi = 0;
    cblock_t *block = NULL;
    cnode_t *node;

    block = list_entry(fn->entry, block, cblock_list);
    list_foreach_entry(&block->cnode_head, node, cnode_list)
    {
        if(node->type != CN_BIND)
            continue;
        locs[node->id] = (loc_t) {
            .base = BASE_ARG,
            .index = argi++
        };
    }
}
```

To minimise copying during SSA deconstruction (Chapter 18) it is desirable that, where possible, a `PHI` node and its arguments are allocated the same concrete location. **phis_assign** connects these 'ϕ-related' nodes with "location hints".

⟨*phis_assign*⟩≡
```
static void phis_assign(cfunction_t *fn, liveranges_t *live, loc_t *locs)
{
    range_t *range;

    list_foreach_entry(&live->range_head, range, range_list)
        if(range->node->type == CN_PHI)
            phi_assign_hints(locs, range->node);
}
```

phi_assign_hints initialises the locations of the `phi` and its arguments. Placed in the array `arr` and sorted in program order (i.e. increasing `.id`), each ϕ-related `node` is examined.

⟨*phi_assign_hints*⟩≡
```
static void phi_assign_hints(loc_t *locs, cnode_t *phi)
{
    cnode_array_t arr = ARRAY_INIT;
    cnode_t *node, *tail = NULL;

    array_copy(&arr, &phi->phi.args);
    array_push(&arr, phi);
    qsort(arr.ptr, arr.length, sizeof(cnode_t *), cmp_node_id);
    array_foreach_entry(&arr, node)
    {
        ⟨assign hint⟩
    }
    array_fini(&arr);
}
```

A sequence of hinted locations forms a chain – with `.base` set to `BASE_HINT`, the `.index` field of each is set to the `.id` of the preceding node.

Only an uninitialised `node` will be assigned, but if this one is already hinted, the chain can continue from it. If this is the first `node`, it begins the chain and stays at `BASE_NONE`. Otherwise, the hint is assigned and the `tail` advanced.

⟨*assign hint*⟩≡
```
loc_t *loc = &locs[node->id];

if(loc->base != BASE_NONE)
{
    if(loc->base == BASE_HINT)
        tail = node;
}
else if(!tail)
    tail = node;
else if(node != tail)
{
    *loc = (loc_t) {
        .base = BASE_HINT,
        .index = tail->id
    };
    tail = node;
}
```

The `qsort` call uses the `cmp_node_id` helper to sort the nodes by `.id`.

⟨*cmp_node_id*⟩≡
```
static int cmp_node_id(const void *a, const void *b)
{
    cnode_t *x = *(cnode_t **)a, *y = *(cnode_t **)b;
    return x->id - y->id;
}
```

17.4 Linear Scan

Each node could be given a separate, unique stack offset, but this would waste space; most values are not live over the entire function. We allocate locations to nodes in a single *linear scan*. This can produce code of nearly the same quality as a conventional graph-colouring register allocator, while being faster and simpler (Traub et al., 1998).

⟨*linear scan*⟩≡
  ⟨*find_free_index*⟩
  ⟨*loc_alloc*⟩

Each storage base tracks the current allocation state with a `local_t` structure. It holds the lists of live ranges that are `.active` and `.inactive`, and an array of flags recording whether each index in the storage base is `.used` by a value.

⟨*local_t*⟩≡
```
typedef struct
{
    list_t active, inactive;
    ARRAY(bool) used;
} local_t;
```

The `loc_alloc` function assigns, to each node, an abstract location for its value that doesn't interfere with any other. The algorithm employed is that of Wimmer and Mössenböck (2005, Figure 2), with insights from Wimmer and Franz (2010, our "live range" corresponding to their "lifetime interval"). It performs linear scan with lifetime holes, but without range splitting or spilling.

⟨*loc_alloc*⟩≡
```
void loc_alloc(cfunction_t *fn, liveranges_t *live, loc_t *locs, unsigned nlocs[])
{
    local_t locals[LOCAL_BASES];
    list_t *unhandled = &live->range_head;
    range_t *cur;

    ⟨allocate locations⟩
}
```

The elements of the `locals` array are first initialised with empty lists and arrays.

⟨*allocate locations*⟩≡
```
for(int i = 0; i < LOCAL_BASES; i++)
{
    local_t *local = &locals[i];

    *local = (local_t) {
        .active = LIST_INIT(local->active),
        .inactive = LIST_INIT(local->inactive),
        .used = ARRAY_INIT
    };
}
```

While there are still nodes with `unhandled` ranges, the range `cur` is taken from the head of the list. This occurs in order of increasing `.start` point, as the list is sorted (Subsection 16.2.1). If the node's location `loc` is not `UNASSIGNED`, its allocation is skipped. The `local` state for the `.node`'s storage `base` is consulted.

⟨*allocate locations*⟩+≡
```
list_while_entry(unhandled, cur, range_list)
{
    range_t *range, *tmp;
    locbase base = base_for(cur->node);
    local_t *local = &locals[base];
    loc_t *loc = &locs[cur->node->id];

    list_remove(&cur->range_list);
    if(loc->base < UNASSIGNED)
        continue;
    ⟨allocate range⟩
}
```

The scan advances to the `.start` position of the range `cur`. Each `.active range` at this storage base is examined. If it has ended, it's removed from the list, and the `.index` that its node occupied is marked as no longer `.used`. If it has a lifetime hole at this position, it's moved to the `.inactive` list, with its node's index also marked unused.

⟨*allocate range*⟩≡
```
list_foreach_entry_safe(&local->active, range, tmp, range_list)
{
    if(range->end <= cur->start)
    {
        list_remove(&range->range_list);
        aset(&local->used, locs[range->node->id].index, false);
    }
    else if(!range_covers(range, cur->start))
    {
        aset(&local->used, locs[range->node->id].index, false);
        list_remove(&range->range_list);
        list_add(&local->inactive, &range->range_list);
    }
}
```

Then each `.inactive range` is examined in a similar manner. If it has has ended, it's removed as well; if it now covers the current position, it's reactivated and its node's index again marked as `.used`.

Note that this is always safe – two values in SSA form only interfere if one is live at the definition of the other (Boissinot et al., 2008, Section III.A), and the end of a lifetime hole is not a definition.

⟨*allocate range*⟩+≡
```
list_foreach_entry_safe(&local->inactive, range, tmp, range_list)
{
    if(range->end <= cur->start)
        list_remove(&range->range_list);
    else if(range_covers(range, cur->start))
    {
        list_remove(&range->range_list);
        list_add(&local->active, &range->range_list);
        aset(&local->used, locs[range->node->id].index, true);
    }
}
```

The unset elements of the `.used` array now accurately reflect the available indices. `find_free_index` returns one; it's marked, the node's location is assigned its `index` and `base`, and the node's range becomes `.active`.

⟨*allocate range*⟩+≡
```
unsigned index = find_free_index(local, locs, loc);

aset(&local->used, index, true);
*loc = (loc_t) {
    .base = base,
    .index = index
};
list_add_before(&local->active, &cur->range_list);
```

After all ranges have been processed, the length of each storage base's `.used` array is equal to the maximum number of values live simultaneously at that base. This is copied to the corresponding element of `nlocs`.

The `local` list heads are unlinked, as they will become invalid when the function returns.

⟨*allocate locations*⟩+≡
```
for(int i = 0; i < LOCAL_BASES; i++)
{
    local_t *local = &locals[i];

    nlocs[i] = alen(&local->used);
    array_fini(&local->used);
    list_remove(&local->inactive);
    list_remove(&local->active);
}
```

The `find_free_index` function returns an available index in the `local` storage base. If the location `loc` was earlier initialised to `BASE_HINT`, it will take the same `.index` as the specified `other` node, if that isn't presently being `.used`.

Otherwise, if an unused index is available, it's returned. The array is extended and the newly added index returned in the event all others are occupied.

⟨*find_free_index*⟩≡
```
static unsigned find_free_index(local_t *local, loc_t *locs, loc_t *loc)
{
    unsigned i;

    if(loc->base == BASE_HINT)
    {
        loc_t *other = &locs[loc->index];

        if(!aref(&local->used, other->index))
            return other->index;
    }
    array_foreach(&local->used, i)
        if(!aref(&local->used, i))
            return i;
    array_extend(&local->used, 1);
    return i;
}
```

As `loc_alloc` doesn't split intervals within a range, a value will never change its location across a control-flow edge. This reduces the resolution phase of Wimmer and Franz (2010, Section 6) to SSA deconstruction only, which occurs in Chapter 18.

17.5 Concrete Locations

The stack frame is subdivided, with an area for each storage base. All local scalar values with the same width are stored together, as are local pointers. This increases overhead compared to an optimal packing, but has a very simple implementation. After the sizes of these areas are computed, abstract locations can be made concrete.

⟨*concrete locations*⟩≡
  ⟨*loc_bases*⟩
  ⟨*arg_offset*⟩
  ⟨*local_offset*⟩
  ⟨*loc_offsets*⟩

`loc_bases` writes the start offset of each local storage base to `base_ofs`.

They are laid out one after the other at the beginning of the stack frame; `BASE_8` followed by `BASE_32` then `BASE_64`. `*pscalsz` is assigned their total size in bytes.

`BASE_PTR` comes after the scalars. Its size is added to the total and the result assigned to `*psz`. Exceeding the maximum addressable local stack space is a fatal error.

⟨*loc_bases*⟩≡
```
static void loc_bases(op_offset_t *base_ofs, unsigned nlocs[],
                      op_offset_t *pscalsz, op_offset_t *psz)
{
    size_t sz = 0;

    for(locbase i = BASE_8; i < SCALAR_BASES; i++)
    {
        base_ofs[i] = sz;
        sz += nlocs[i] * size_base(i);
    }
    *pscalsz = sz;
    if(sz > SHRT_MAX)
        fatal("out of local scalar stack space.");
    base_ofs[BASE_PTR] = sz;
    sz += nlocs[BASE_PTR] * size_base(BASE_PTR);
    if(sz > SHRT_MAX)
        fatal("out of local stack space.");
    *psz = sz;
}
```

Finally, `loc_offsets` can compute the concrete stack location, as an offset in bytes from the VM base pointer, from the abstract location `loc` assigned to each `node`. They are returned in the `ofs` array.

⟨*loc_offsets*⟩≡
```
static op_stack_t *loc_offsets(cfunction_t *fn, op_offset_t *base_ofs,
                               loc_t *locs)
{
    op_stack_t *ofs = xcalloc(fn->nnodes, sizeof(*ofs));
    cblock_t *block;

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        cnode_t *node;

        list_foreach_entry(&block->cnode_head, node, cnode_list)
        {
            loc_t *loc = &locs[node->id];

            if(!cnode_yields_value(node))
                continue;
            ofs[node->id] = (loc->base == BASE_ARG)
                          ? arg_offset(fn, loc)
                          : local_offset(base_ofs, loc);
        }
    }
    return ofs;
}
```

`arg_offset` returns the offset for the function `fn`'s argument with the given `index` at `BASE_ARG`.

The first, with index `0`, is always the implicit `argbits` argument (Chapter 22). The `.offset`s of the remainder have already been computed, and are specified by the function `fn`'s signature (Section 21.2).

⟨*arg_offset*⟩≡
```
static inline op_stack_t arg_offset(cfunction_t *fn, loc_t *loc)
{
    funsig_t *sig = fn->cl_type->sig;

    return -(sig->argsz + sizeof(vm_act_rec_t)) +
            (!loc->index ? 0 : sig->args[loc->index - 1].offset);
}
```

`local_offset` returns the offset in the stack frame for the node with the given `index` at `base`.

⟨*local_offset*⟩≡
```
static inline op_stack_t local_offset(op_offset_t *base_ofs, loc_t *loc)
    { return base_ofs[loc->base] + size_base(loc->base) * loc->index; }
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ir.h"
// for vm_act_rec_t
#include "vm/vm_ops.h"
#include "gen.h"
#include "gen_range.h"
```

⟨*gen.h*⟩≡
```
⟨locbase⟩
⟨loc_t⟩
⟨base_for⟩
⟨size_base⟩
```

Chapter 18

SSA Form Destruction

The ϕ-functions of static single assignment form are not part of the VM instruction set. They are removed, before generating code, by inserting copies (Cytron et al., 1991, Section 7).

⟨*gen_unssa.c*⟩≡
  ⟨*includes*⟩
  ⟨*make_copy*⟩
  ⟨*mapelt_t*⟩
  ⟨*schedmap_t*⟩
  ⟨*map_get*⟩
  ⟨*map_init*⟩
  ⟨*schedule_init*⟩
  ⟨*schedule_ready*⟩
  ⟨*schedule_todo*⟩
  ⟨*schedule_copies*⟩
  ⟨*gen_un_ssa*⟩

This is performed by the `gen_un_ssa` function, corresponding to Wimmer and Franz (2010, Figure 7), or the first part of Briggs et al. (1998, Figure 14, "Pass One").

For each `block` in function `fn`, each `PHI node` in each successor `succ` is considered.

⟨*gen_un_ssa*⟩≡
```
void gen_un_ssa(cfunction_t *fn, op_offset_t temploc)
{
    cblock_t *block;
    list_t copies = LIST_INIT(copies);

    list_foreach_entry(&fn->cblock_head, block, cblock_list)
    {
        gen_block_t *rec = rec_for(block);
        int i, ncopies = 0;

        rec->copies = (list_t) LIST_INIT(rec->copies);
        array_foreach(&block->succ, i)
        {
            cblock_t *succ = aref(&block->succ, i);
            int j = index_of_block(&succ->pred, block);
            cnode_t *node;

            list_foreach_entry(&succ->cnode_head, node, cnode_list)
            {
                if(node->type != CN_PHI)
                    break;
                ⟨copy for phi arg⟩
```

```
                }
            }
            ⟨schedule copies⟩
        }
    }
```

The argument `arg` corresponds to the value of the `PHI` when control-flow enters `succ` from `block`. Locations on the VM stack have been allocated to each node (Chapter 17). If the argument and the `PHI` itself were given different locations, a `copy` from the former to the latter is required.

⟨*copy for phi arg*⟩≡
```
cnode_t *arg = aref(&node->phi.args, j);
op_stack_t dest = loc_for(node), src = loc_for(arg);

if(src != dest)
{
    gen_copy_t *copy = make_copy(dest, src, decl_type(node->decl));

    list_add(&copies, &copy->list);
    ncopies++;
}
```

These `copies`, performed in parallel, faithfully implement the semantics of the ϕ-function, avoiding the "swap problem" of Briggs et al. (1998, Subsection 5.1). The "lost-copy problem" does not occur, as critical edges have been split.

`schedule_copies` finds an equivalent sequential ordering for the copies, possibly requiring an additional temporary location `temploc`. Its output is placed in the `.copies` field of the `block`'s record `rec`, for use during code generation (Section 15.4).

⟨*schedule copies*⟩≡
```
if(ncopies > 0)
    schedule_copies(temploc, &copies, ncopies, &rec->copies);
```

The `make_copy` helper creates and returns a request to `copy` a value of certain `type` between the given locations.

⟨*make_copy*⟩≡
```
static gen_copy_t *make_copy(op_stack_t dest, op_stack_t src, rtype_t *type)
{
    gen_copy_t *copy = xmalloc(sizeof(*copy));

    *copy = (gen_copy_t) {
        .dest = dest,
        .src = src,
        .type = type
    };
    return copy;
}
```

18.1 Copy Scheduling

The `schedmap_t` structure subsumes the loc and pred arrays of Boissinot et al. (2008, Algorithm 1), as stack locations are sparse. It could be replaced with a hash table or equivalent container with sublinear access time, if large numbers of entries are expected.

⟨*schedmap_t*⟩≡
```
typedef struct {
    int nelts;
    mapelt_t *elts;
} schedmap_t;
```

The `mapelt_t` entry with `.el` equal to k stores $\mathrm{loc}(k)$ in its `.loc` field. Its `.pred` field references the (unique) copy with `.dest` equal to $\mathrm{pred}(k)$.

⟨*mapelt_t*⟩≡
```
typedef struct {
    op_stack_t el, loc;
    gen_copy_t *pred;
} mapelt_t;
```

The `map_get` helper returns the `map` element `elt` corresponding to the stack location `el`.

⟨*map_get*⟩≡
```
static inline mapelt_t *map_get(schedmap_t *map, op_stack_t el)
{
    for(int i = 0; i < map->nelts; i++)
    {
        mapelt_t *elt = &map->elts[i];
        if(elt->el == el)
            return elt;
    }
    return NULL;
}
```

The `map_init` helper specifies that the initial location of `map` element `i` is at stack location `el`.

⟨*map_init*⟩≡
```
static inline void map_init(schedmap_t *map, int i, op_stack_t el)
{
    map->elts[i].el = el;
    map->elts[i].loc = el;
}
```

`schedule_copies` sequentializes the copies on the `input` list, possibly with additional copies via `temploc`. The `map` contains one element per copy, and one for the temporary. A copy yet to be scheduled is linked into one of the `ready` and `todo` lists (a difference from algorithm 1, where it can appear on both.)

After these are populated by `schedule_init` (algorithm 1, lines 1-9), the loops repeat until all copies have been moved to the `output` list – unblocked copies from the `ready` list by `schedule_ready` (lines 12-16), and blocked ones from `todo` by `schedule_todo` (lines 17-21).

⟨*schedule_copies*⟩≡
```
static void schedule_copies(op_offset_t temploc, list_t *in, unsigned ncopies,
                            list_t *out)
{
    list_t ready = LIST_INIT(ready), todo = LIST_INIT(todo);
    schedmap_t map = {
        .nelts = ncopies + 1,
        .elts = xcalloc(ncopies + 1, sizeof(mapelt_t))
    };

    schedule_init(in, temploc, &map, &ready, &todo);
    while(!list_isempty(&ready) || !list_isempty(&todo))
    {
        while(!list_isempty(&ready))
            schedule_ready(container_of(ready.next, gen_copy_t, list),
                           out, &map, &ready);
        if(!list_isempty(&todo))
            schedule_todo(container_of(todo.next, gen_copy_t, list),
                          temploc, out, &map, &ready);
    }
    xfree(map.elts);
}
```

`schedule_init` begins by adding an element to the `map` for each copy in the `input`, and for the temporary.

If the `.dest` of some `copy` coincides with the `.src` of another, the former is *blocked* – it must wait for the latter to copy the value at that location away before it can safely execute. The blocked copy is recorded in the `.pred` field of the element at `.dest`, and is linked into the deferred `todo` list.

An unblocked copy may be scheduled immediately, and is linked into the `ready` list.

⟨*schedule_init*⟩≡
```
static inline void schedule_init(list_t *in, op_stack_t temploc, schedmap_t *map,
                                 list_t *ready, list_t *todo)
{
    gen_copy_t *copy, *tmp;
    int i = 0;

    list_foreach_entry(in, copy, list)
        map_init(map, i++, copy->src);
    map_init(map, i, temploc);

    list_foreach_entry_safe(in, copy, tmp, list)
    {
        mapelt_t *elt = map_get(map, copy->dest);

        if(elt)
        {
            elt->pred = copy;
```

```
                list_remove(&copy->list);
                list_add(todo, &copy->list);
            }
            else
            {
                list_remove(&copy->list);
                list_add(ready, &copy->list);
            }
        }
    }
```

A `copy` from the `ready` list is moved to the `output` by `schedule_ready`. The `map` element `elt` for the copy's `.src` location is examined. If the value is still there, blocking another copy `.pred`, the latter can now be made `ready`. The fields of copy and element are updated – the value which began at `.src` has been copied, possibly via `.loc`, to `.dest`.

⟨*schedule_ready*⟩≡

```
static void schedule_ready(gen_copy_t *copy, list_t *out, schedmap_t *map,
                           list_t *ready)
{
    mapelt_t *elt = map_get(map, copy->src);

    list_remove(&copy->list);
    list_add_before(out, &copy->list);
    if(copy->src == elt->loc && elt->pred)
    {
        list_remove(&elt->pred->list);
        list_add(ready, &elt->pred->list);
    }
    copy->src = elt->loc;
    elt->loc = copy->dest;
}
```

A `copy` which remains on the `todo` list belongs to a cycle of blocked copies. Adding a copy from `.src` to a temporary location `temploc` breaks the cycle. The corresponding `map` element `elt` has its `.loc` field updated to reflect the value's new location. Now unblocked, the `.pred`ecessor copy can be made `ready`. The `copy` itself will be handled after it's unblocked in `schedule_ready`.

⟨*schedule_todo*⟩≡

```
static void schedule_todo(gen_copy_t *copy, op_offset_t temploc, list_t *out,
                          schedmap_t *map, list_t *ready)
{
    mapelt_t *elt = map_get(map, copy->src);
    gen_copy_t *tempcopy = make_copy(temploc, copy->src, copy->type);

    elt->loc = temploc;
    list_add_before(out, &tempcopy->list);
    list_remove(&elt->pred->list);
    list_add(ready, &elt->pred->list);
    list_remove(&copy->list);
}
```

Miscellanea

⟨*includes*⟩≡

```
#include "global.h"
#include "ir.h"
#include "gen.h"
// for gen_code.h
#include "vm/vm_ops.h"
// for the globals and loc_for/rec_for
#include "gen_code.h"
```

Part II

Run-Time Environment

Chapter 19

Runtime

The runtime subsystem offers a set of data types and objects to user code, and mediates with the operating system. It defines the behaviour of these types and objects (Chapters 20–24), manages the global environments, and handles memory allocation and garbage collection (Chapter 25).

⟨*rt/runtime.c*⟩≡
  ⟨*includes*⟩
  ⟨*ops*⟩
  ⟨*runtime_root*⟩
  ⟨*nil*⟩
  ⟨*symbols*⟩
  ⟨*strings*⟩
  ⟨*cells*⟩
  ⟨*global variables*⟩
  ⟨*runtime_init*⟩
  ⟨*runtime_fini*⟩

The runtime module orchestrates initialisation and finalisation of the subsystem, manages global variables, interfaces with the garbage collector, and defines the primitive types and values not belonging to other modules – objects, symbols, strings, cells, and `nil`.

19.1 Objects

Values manipulated by user code are pointers to `robject_t`. Every type of object includes one of these structures as its first member. Its `.type` field points to the unique `rtype_t` which describes the set of instances to which the object belongs (Chapter 20).

⟨*robject_t*⟩≡

```
typedef struct rtype rtype_t;
typedef struct
{
    rtype_t *type;
} robject_t;
typedef ARRAY(robject_t *) robject_array_t;
```

The compiler may decide that a value can be instead be represented as a *scalar* of some type handled by the underlying processor.

We use these `typedef`s to remind ourselves (and `gcc`) that runtime scalars are not C scalars, as each of the former has a designated `NA` value.

When a scalar is handled by code which has not been specialised by the compiler, it's placed in a *box*, and the pointer to it is treated like any other object (Subsection 23.1.1).

⟨*scalar types*⟩≡
```
typedef double rdouble_t;
typedef int rint_t;
typedef uint8_t rboolean_t;
```

Templated code in the VM is generically applicable to scalars and objects. This `typedef` allows the type name "ptr" to be used in such contexts.

⟨*pointer type*⟩≡
```
typedef robject_t *rptr_t;
#define rptr_na NULL
```

Some fundamental functions may be called on objects of any type. These are mostly used by the compiler and runtime (but may, via builtins, be called from user code.)

To implement type-specific behaviour, these functions rely on the callbacks specified in the `.ops` field of an object's `.type`.

⟨*typeops_t*⟩≡
```
typedef struct
{
    uint32_t (*hash)(const void *);
    bool (*equal)(const void *, const void *);
    void (*print)(FILE *, const void *);
    void (*gc)(void *);
    void (*free)(void *);
} typeops_t;
```

The arguments are typed as `void *` so they can be invoked on specialised objects (which contain, but are not themselves, `robject_t`) without being obliged to insert casts.

Given an object, `r_typeof` returns its type. The `nil` object is represented as a C `NULL` pointer, and has all bits zero. (not strictly standards compliant.) Its type is distinct, `r_type_nil`.

⟨*r_typeof*⟩≡
```
static inline rtype_t *r_typeof(const void *ptr)
    { return ptr ? ((robject_t *)ptr)->type : r_type_nil; }
```

19.1.1 Primitive Operations

⟨*ops*⟩≡
```
⟨r_hash⟩
⟨r_equal⟩
⟨r_print⟩
⟨r_gc⟩
⟨r_free⟩
```

r_hash computes the 32-bit hash of the object. This value differs (with high probability) for objects which are not equal. Since the notion of equality differs depending on the object's type, we invoke its ops.hash callback. All concrete types (i.e. those which might be returned by r_typeof called on some object) are expected to populate this field.

⟨*r_hash*⟩≡
```
uint32_t r_hash(const void *ptr)
{
    rtype_t *typ = r_typeof(ptr);
    assert(typ->ops->hash);
    return typ->ops->hash(ptr);
}
```

hash_roll is used to mix two hashes together. The hash of a composite object can be computed by "accumulating" the value hash = hash_roll(hash, r_hash(element)) for each element.

⟨*hash_roll*⟩≡
```
static inline uint32_t hash_roll(uint32_t hash, uint32_t val)
    { return hash_code_seed(&val, sizeof(val), hash); }
```

Since objects are represented by pointers, and an object is always equal to itself, we can short-circuit the ops.equal call if the pointers are identical. Equality is symmetric – equal(a,b) == equal(b,a) – and objects of different types are never equal. As a convenience, omitting the callback denotes that every object of that type is unique.

⟨*r_equal*⟩≡
```
bool r_equal(const void *xp, const void *yp)
{
    const robject_t *x = xp, *y = yp;
    rtype_t *xt = r_typeof(x), *yt = r_typeof(y);

    if(x == y)
        return true;
    if(xt != yt)
        return false;
    if(xt->ops->equal)
        return xt->ops->equal(xp, yp);
    return false;
}
```

Output from r_print is intended for human interpretation. The FILE * argument is usually stdout, but explicitly allows for programmatic redirection to a file or memory stream. If the object's type doesn't provide an ops.print callback, some indication of object identity is output instead, in the form "<type pointer>".

No special care is taken, when printing objects, to recognise and avoid cycles in the reference graph. An infinite loop will result from printing an object that directly or indirectly contains a pointer to itself.

⟨*r_print*⟩≡
```
void r_print(FILE *fp, const void *ptr)
{
    rtype_t *typ = r_typeof(ptr);

    if(typ->ops->print)
        typ->ops->print(fp, ptr);
    else
    {
        fprintf(fp, "<");
```

```
        r_print(fp, typ);
        fprintf(fp, " %p>", ptr);
    }
}
```

The garbage collector (Chapter 25) may call `r_gc` on an object. It is responsible for calling `gc_mark` on any objects it holds references to – so, at least its `.type`. If it has any others, they will be marked by `ops.gc`.

⟨*r_gc*⟩≡
```
void r_gc(void *ptr)
{
    rtype_t *typ = r_typeof(ptr);

    gc_mark(typ);
    if(typ->ops->gc)
        typ->ops->gc(ptr);
}
```

If an object is not marked during collection, it will never be used again, and will be swept as garbage when the collector calls `r_free`. Some types of objects contain pointers to unmanaged memory (i.e. not `robject_t`s,) and these will be deallocated by `ops.free`.

⟨*r_free*⟩≡
```
void r_free(void *ptr)
{
    rtype_t *typ = r_typeof(ptr);
    if(typ->ops->free)
        typ->ops->free(ptr);
}
```

Some objects may not be referred to by other objects but are accessible nevertheless, and so are not garbage. The collector keeps a list of `gc_root_t`s, and invokes each `.fn` to mark these objects.

⟨*runtime_root*⟩≡
```
⟨mark_global⟩
⟨mark_type⟩
⟨runtime_gc⟩
static gc_root_t runtime_root = { .fn = runtime_gc };
```

The runtime marks the variables and types in the global environments, as well as some implementation details which are not exposed to the user (the mutable cell and compiled function types, and the `...` symbol)

⟨*runtime_gc*⟩≡
```
static void runtime_gc(gc_root_t *root)
{
    hashmap_map(r_globals, mark_global, NULL);
    hashmap_map(r_global_types, mark_type, NULL);
    gc_mark(r_type_cell);
    gc_mark(r_type_function);
    gc_mark(r_sym_rest);
}
```

Each global variable has its name and declared type marked. Its value is also marked if it's an object and not an unboxed scalar.

⟨*mark_global*⟩≡

```
static void mark_global(const void *key, void *value, void *data)
{
    rglobal_t *global = value;

    gc_mark((void *)key);
    gc_mark(global->decl);
    if(!global->decl || !rtype_is_scalar(global->decl))
        gc_mark(global->val.object);
}
```

Each type in the global environment is marked, as is its name.

⟨*mark_type*⟩≡

```
static void mark_type(const void *key, void *value, void *data)
{
    gc_mark((void *)key);
    gc_mark(value);
}
```

19.1.2 nil

`r_type_nil` has its own `ops.print` and `ops.hash` callbacks.

⟨*nil*⟩≡

```
static uint32_t nil_hash(const void *ptr)
{
    return 0;
}

static void nil_print(FILE *fp, const void *ptr)
{
    fprintf(fp, "<nil>");
}

static const typeops_t nil_ops = {
    .hash = nil_hash,
    .print = nil_print
};
```

19.2 Symbols

A symbol is an immutable unique string, as in LISP (McCarthy, 1960). Only one symbol with a given string is stored in the symbol table, so (except in `r_intern`) two symbols may be compared with pointer equality in constant time.

⟨*symbols*⟩≡

    ⟨*r_symtab*⟩
    ⟨*sym_init*⟩
    ⟨*sym_equal*⟩
    ⟨*rsym_hash*⟩
    ⟨*r_intern*⟩
    ⟨*sym_free*⟩
    ⟨*sym_print*⟩
    ⟨*sym_ops*⟩

The `.string` field points to `libc` heap memory owned by the object.

⟨*rsymbol_t*⟩≡
```
typedef struct rsymbol
{
    robject_t base;
    char *string;
    uint32_t hash;
} rsymbol_t;
```

The `.hash` of the string is precomputed and passed in during initialisation.

⟨*sym_init*⟩≡
```
static void sym_init(rsymbol_t *sym, char *string, uint32_t hash)
{
    sym->string = string;
    sym->hash = hash;
}
```

When their `.string`s are equal, two symbols are equal.

⟨*sym_equal*⟩≡
```
static bool sym_equal(const void *xp, const void *yp)
{
    const rsymbol_t *x = xp, *y = yp;
    return string_equal(x->string, y->string);
}
```

The `.hash` of a symbol can be extracted.

⟨*rsym_hash*⟩≡
```
uint32_t rsym_hash(const void *ptr)
{
    return ((rsymbol_t *)ptr)->hash;
}
```

`nil` is a member of `r_type_symbol`, and has a distinguished representation.

⟨*r_symstr*⟩≡
```
static inline char *r_symstr(const rsymbol_t *sym)
    { return sym ? sym->string : "(nil)"; }
```

All symbols are stored in the symbol table, a hash set whose elements are compared with `sym_equal` and hashed with `rsym_hash`.

⟨*r_symtab*⟩≡
```
hashset_t *r_symtab;
```

A symbol is created by "internalising" a string. The string is hashed, and the symbol table is probed with a temporary `rsymbol_t`. If some symbol is found, it is returned. Otherwise a new symbol is allocated, initialised, and returned. The `string` argument is duplicated, so the caller may reuse its copy.

⟨*r_intern*⟩≡
```
rsymbol_t *r_intern(const char *string)
{
    assert(string);
    rsymbol_t *sym, tmp;
    uint32_t hash = string_hash(string);

    sym_init(&tmp, (char *)string, hash);
    sym = hashset_get(r_symtab, &tmp);
    if(sym)
        return sym;
```

```
    sym = gc_alloc(r_type_symbol, sizeof(rsymbol_t));
    sym_init(sym, strdup(string), hash);
    hashset_insert(r_symtab, sym);
    return sym;
}
```

Creating symbols is only possible via `r_intern`, so `r_symtab` contains all currently live symbols. When a symbol is not reachable from any root, and is collected as garbage, it may be safely removed from the table.

⟨*sym_free*⟩≡

```
static void sym_free(void *ptr)
{
    rsymbol_t *sym = ptr;

    hashset_remove(r_symtab, sym);
    xfree(sym->string);
}
```

A symbol is printed as its `.string` (or `nil`,) without decoration.

⟨*sym_print*⟩≡

```
static void sym_print(FILE *fp, const void *ptr)
{
    fprintf(fp, "%s", r_symstr(ptr));
}
```

The `.equal` callback defaults to pointer equality. The only user of `sym_equal` is the interning mechanism.

⟨*sym_ops*⟩≡

```
static const typeops_t sym_ops = {
    .free = sym_free,
    .hash = rsym_hash,
    .equal = NULL,
    .print = sym_print
};
```

19.3 Strings

A string is a sequence of characters. It is immutable, but not unique. Strings are very simple, and only really suitable for filenames, error messages or fixed output. An enhancement would be to replace this type with one more fully-featured, possibly a vector (Chapter 24) of Unicode code points.

⟨*strings*⟩≡
⟨*rstr_create*⟩
⟨*rstr_free*⟩
⟨*rstr_equal*⟩
⟨*rstr_hash*⟩
⟨*rstr_print*⟩
⟨*str_ops*⟩

As with `r_type_symbol`, the heap memory pointed to by the `.string` field is owned by the object.

⟨*rstring_t*⟩≡

```
typedef struct
{
    robject_t base;
    char *string;
} rstring_t;
```

Creation is straightforward. The `string` argument is duplicated.

⟨*rstr_create*⟩≡
```
rstring_t *rstr_create(const char *string)
{
    rstring_t *str = gc_alloc(r_type_string, sizeof(rstring_t));
    str->string = strdup(string);
    return str;
}
```

Destruction is likewise trivial.

⟨*rstr_free*⟩≡
```
static void rstr_free(void *ptr)
{
    rstring_t *str = ptr;
    xfree(str->string);
}
```

`rstr_equal` is (mutatis mutandis) identical to `rsym_equal`.

⟨*rstr_equal*⟩≡
```
static bool rstr_equal(const void *xp, const void *yp)
{
    const rstring_t *x = xp, *y = yp;
    return string_equal(x->string, y->string);
}
```

Unlike symbols, strings are not expected to be hashed often, so they do not store a `.hash` field but compute it each time.

⟨*rstr_hash*⟩≡
```
static uint32_t rstr_hash(const void *ptr)
{
    const rstring_t *str = ptr;
    return string_hash(str->string);
}
```

A string is printed inside `"double quotes"`.

⟨*rstr_print*⟩≡
```
static void rstr_print(FILE *fp, const void *ptr)
{
    const rstring_t *str = ptr;
    fprintf(fp, "\"%s\"", str->string);
}
```

Note the `.equal` callback is initialised – unlike symbols, two strings with the same contents are not necessarily the same object.

⟨*str_ops*⟩≡
```
static const typeops_t str_ops = {
    .free = rstr_free,
    .hash = rstr_hash,
    .equal = rstr_equal,
    .print = rstr_print
};
```

19.4 Cells

The values of mutable variables captured by a function are not stored directly in its closure. *Cells* are introduced by the compiler to hold their values (Chapter 12). Many closures may refer to the same cell.

⟨*cells*⟩≡
  ⟨*rcell_create*⟩
  ⟨*rcell_gc*⟩
  ⟨*cell_ops*⟩
  ⟨*cell_cons*⟩
  ⟨*rcell_type_create*⟩

A cell, like a box, contains a single value of its element type, stored immediately following the `robject_t` header.

⟨*rcell_create*⟩≡
```
robject_t *rcell_create(rtype_t *typ)
{
    assert(rtype_is_cell(typ));
    return gc_alloc(typ, sizeof(robject_t) + rtype_eltsz(typ->elt));
}
```

If its element isn't an unboxed scalar, it's a reference to an object, and must be marked during garbage collection.

⟨*rcell_gc*⟩≡
```
static void rcell_gc(void *ptr)
{
    robject_t *cell = ptr;
    rtype_t *etyp = r_typeof(cell)->elt;

    if(!rtype_is_scalar(etyp))
        gc_mark(UNBOX(robject_t *, cell));
}
```

Cells aren't visible to the user and allocate no unmanaged memory, so need no other `.ops` callbacks.

⟨*cell_ops*⟩≡
```
static const typeops_t cell_ops = {
    .gc = rcell_gc,
};
```

`r_type_cell` is a type constructor, but it can't be invoked by the user so `.from_spec` need not be provided.

⟨*cell_cons*⟩≡
```
static consdesc_t cell_cons = {
    .kind = RT_CELL
};
```

The compiler calls `rcell_type_create` to create the type instance of a cell holding an element of type `etype`.

⟨*rcell_type_create*⟩≡
```
rtype_t *rcell_type_create(rtype_t *etyp)
{
    return rtype_create(RT_CELL, etyp, &cell_ops, NULL);
}
```

19.5 Globals

The runtime defines variables in the global environment to provide objects to the user code (which may define variables of its own.)

⟨*global variables*⟩≡
  ⟨*r_globals*⟩
  ⟨*r_get_global*⟩
  ⟨*r_create_global*⟩
  ⟨*r_unset*⟩
  ⟨*r_definitial*⟩

A global variable is represented by an `rglobal_t` binding structure. If `.decl` is `NULL`, the global was not explicitly declared by the user. Otherwise it points to an object which bounds the variable's type – it will always be the case that `r_subtypep(.val, .decl)`. `.is_const` is set if the global was declared constant – its `.val` will never change (note that this is not a property of the object to which it refers, e.g. the elements of a vector or array may still be modified.)

⟨*rglobal_t*⟩≡
```
typedef struct
{
    rvalue_union_t val;
    rtype_t *decl;
    bool is_const;
} rglobal_t;
```

The value of a global variable is stored in an `rvalue_union_t`. When specified by `.decl`, this may contain a scalar so a box doesn't need to be allocated. Otherwise (or when the global is undeclared), the value is referred to by the `.object` field.

⟨*rvalue_union_t*⟩≡
```
typedef union {
    rboolean_t boolean;
    rint_t integer;
    rdouble_t dfloat;
    robject_t *object;
} rvalue_union_t;
```

The global environment is a hash table mapping `rsymbol_t` names to `rglobal_t` bindings.

⟨*r_globals*⟩≡
```
hashmap_t *r_globals;
```

If no global exists with the given name, `r_get_global` returns `NULL`.

⟨*r_get_global*⟩≡
```
rglobal_t *r_get_global(rsymbol_t *name)
{
    return hashmap_get(r_globals, name, NULL);
}
```

A global is added to the environment on creation. Checking for an existing variable with the same name is the caller's responsibility (as, in some cases, its absence can be assumed.)

⟨*r_create_global*⟩≡
```
rglobal_t *r_create_global(rsymbol_t *name, rtype_t *decl, bool is_const)
{
    rglobal_t *global = xcalloc(1, sizeof(rglobal_t));

    assert(!r_get_global(name));
    *global = (rglobal_t) {
        .decl = decl,
        .is_const = is_const
    };
    hashmap_set(r_globals, name, global);
    return global;
}
```

Declaration of a global guarantees its existence with unchanging `.decl`, `.is_const` and, if the latter, `.val`. The compiler can then safely take advantage of these properties when generating code.

Only globals which have not been explicitly declared may be removed from the environment.

⟨*r_unset*⟩≡
```
bool r_unset(rsymbol_t *name)
{
    rglobal_t *global = r_get_global(name);

    if(global && !global->decl)
    {
        hashmap_remove(r_globals, name);
        xfree(global);
        return true;
    }
    return false;
}
```

Runtime modules use `r_definitial` to create a global constant with an initial value. `pval` points to an object of the type `decl`.

⟨*r_definitial*⟩≡
```
void r_definitial(rsymbol_t *name, rtype_t *decl, void *pval)
{
    assert(decl);
    rglobal_t *global = r_create_global(name, decl, true);
    memcpy(&global->val, pval, rtype_eltsz(decl));
}
```

Convenience wrappers initialise global constants with values of reference types…

⟨*r_defbuiltin*⟩≡
```
static inline void r_defbuiltin(char *str, void *value)
    { r_definitial(r_intern(str), r_typeof(value), &value); }
```

…and scalar types, respectively.

⟨*r_defscalar*⟩≡
```
static inline void r_defscalar(char *str, rtype_t *decl,
                               rvalue_union_t value)
    { r_definitial(r_intern(str), decl, &value); }
```

19.6 Initialisation and Finalisation

The runtime environment has process lifetime. It is the first module to be brought up, and the last to be shut down.

⟨runtime_init⟩≡
```
rtype_t *r_type_nil, *r_type_symbol, *r_type_string;
rtype_t *r_type_cell, *r_type_object;
rsymbol_t *r_sym_rest;
const extern builtin_init_t runtime_builtins[];
const extern builtin_init_t arith_builtins[];
void runtime_init()
{
    ⟨create tables⟩
    ⟨init gc⟩
    ⟨init vm⟩
    ⟨bootstrap type⟩
    ⟨install runtime types⟩
    ⟨install types and builtins⟩
}
```

The global environment and symbol table are allocated first (global bindings are unique, so `ptr_eq` suffices as equality predicate.)

⟨create tables⟩≡
```
r_globals = hashmap_create(rsym_hash, ptr_eq);
r_symtab = hashset_create(rsym_hash, sym_equal);
```

The managed heap must be initialised before any managed objects are created. `runtime_root` is registered with the collector, but garbage collection is not enabled until we specify otherwise – we need not worry about unrooted or partially initialised objects being collected out from under us.

⟨init gc⟩≡
```
gc_init();
gc_register(&runtime_root);
```

The VM must export its direct-threading jump table before any bytecode can be generated by the compiler (Section 26.3).

⟨init vm⟩≡
```
vm_execute(NULL, NULL);
```

`r_type_type` is created (carefully) by `rt_bootstrap`. Once that exists, we can create new types in the usual manner. However, before we can install them in the global type environment, we must initialise `r_type_symbol`, lest we confuse `r_intern`.

⟨bootstrap type⟩≡
```
rt_bootstrap();
r_type_symbol = rtype_create(RT_OBJECT, NULL, &sym_ops, "symbol");
rtype_install(r_type_type, "type");
rtype_install(r_type_symbol, "symbol");
```

`r_type_object` is "abstract", in the sense that no objects exist with it as their `.type`. Its `.ops` will never be invoked, so may be `NULL`.

The rest of the runtime types are created and, except for `r_type_cell`, installed (the latter is an implementation detail of closures, and may only appear in compiled code.)

The `nil` constant is defined here – as is the symbol `...`, as builtins defined later may use it as an argument name.

⟨*install runtime types*⟩≡
```
r_type_object = rtype_init(RT_OBJECT, NULL, "object");
r_type_nil = rtype_init(RT_OBJECT, &nil_ops, "nil");
r_type_string = rtype_init(RT_OBJECT, &str_ops, "string");
r_type_cell = rtype_cons_create(&cell_cons, "cell");
r_defbuiltin("nil", NULL);
r_sym_rest = r_intern("...");
```

Other modules may then proceed to define types, objects and builtin functions in the global environment(s).

⟨*install types and builtins*⟩≡
```
rt_install_call_types();
rt_install_scalar_types();
rt_install_vec_types();
rbuiltin_install(arith_builtins);
rbuiltin_install(runtime_builtins);
rt_init_options();
```

Finalisation is simpler: since most objects are managed, shutting down the heap will destroy them. Global variable bindings aren't (neither are the global tables), so they must be explicitly freed.

⟨*runtime_fini*⟩≡
```
static void free_global(const void *key, void *value, void *data)
{
    xfree(value);
}

void runtime_fini()
{
    gc_fini();
    hashmap_map(r_globals, free_global, NULL);
    hashmap_free(r_globals);
    hashset_free(r_symtab);
    hashset_free(r_typetab);
    hashmap_free(r_global_types);
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "builtin.h"
```

⟨*rt/runtime.h*⟩≡
⟨*scalar types*⟩
⟨*robject_t*⟩
⟨*pointer type*⟩
⟨*typeops_t*⟩
⟨*rsymbol_t*⟩
⟨*rstring_t*⟩
⟨*rvalue_union_t*⟩
⟨*rglobal_t*⟩
⟨*externs*⟩
⟨*prototypes*⟩
⟨*RTYPE*⟩
⟨*hash_roll*⟩
⟨*r_symstr*⟩
⟨*r_typeof*⟩
⟨*r_defbuiltin*⟩
⟨*r_defscalar*⟩
⟨*subincludes*⟩

⟨*RTYPE*⟩≡
```
#define RTYPE(type) PASTE2(r_type_, type)
```

⟨*externs*⟩≡
```
extern hashmap_t *r_globals;
extern hashset_t *r_symtab;
extern rsymbol_t *r_sym_rest;
extern rtype_t *r_type_object, *r_type_symbol, *r_type_string;
extern rtype_t *r_type_nil, *r_type_cell;
```

⟨*prototypes*⟩≡
```
uint32_t rsym_hash(const void *sym);
rsymbol_t *r_intern(const char *string);
rstring_t *rstr_create(const char *string);
void r_free(void *ptr);
void r_gc(void *ptr);
uint32_t r_hash(const void *ptr);
bool r_equal(const void *px, const void *py);
void r_print(FILE *fp, const void *ptr);
rglobal_t *r_get_global(rsymbol_t *name);
rglobal_t *r_create_global(rsymbol_t *name, rtype_t *decl, bool is_const);
bool r_unset(rsymbol_t *name);
void r_definitial(rsymbol_t *name, rtype_t *decl, void *pval);
rtype_t *rcell_type_create(rtype_t *etyp);
robject_t *rcell_create(rtype_t *etyp);
void runtime_init();
void runtime_fini();
```

⟨*subincludes*⟩≡
```
#include "rt/type.h"
#include "rt/scalar.h"
#include "rt/callable.h"
#include "rt/call.h"
#include "rt/vector.h"
#include "rt/gc.h"
#include "rt/math.h"
#include "rt/options.h"
#include "vm/vm.h"
```

Chapter 20

Types

Every object belongs to a *type*. This describes its layout in memory, and determines the behaviour of the functions to which it may be passed.

⟨*rt/type.c*⟩≡
  ⟨*includes*⟩
  ⟨*types*⟩
  ⟨*subtyping*⟩
  ⟨*creation*⟩
  ⟨*constructors*⟩
  ⟨*rt_bootstrap*⟩

`rtype_t`s are referred to at run-time by objects, and at compile-time by expressions and variables as they are declared by the user or recovered by the compiler.

20.1 Types and Kinds

A type is an object, so it begins with an included `robject_t` structure. Its `.hash` is precomputed, like a symbol's.

The callbacks in a type's `.ops` implement the behaviour of the associated runtime functions when they are invoked on an object of the given type.

The `.name` field contains a printable representation of the type – for some kinds of type this will be lazily initialised, and may be `NULL`.

⟨*rtype_t*⟩≡

```
typedef struct funsig funsig_t;
typedef struct scaldesc scaldesc_t;
typedef struct rtype
{
    robject_t base;
    type_kind kind;
    uint32_t hash;
    const typeops_t *ops;
    char *name;
    union
    {
        void *data;
        funsig_t *sig;
        const scaldesc_t *scal;
        struct rtype *elt;
        const consdesc_t *cons;
    };
} rtype_t;
```

A type might belong to a family of types which are related. These families are called *kinds* for disambiguation, and the kind of a type is recorded in its `.kind` field.

⟨*type_kind*⟩≡
```
typedef enum
{
    RT_OBJECT, RT_SCALAR, RT_TYPECONS,
    RT_CALLABLE, RT_VECTOR, RT_ARRAY, RT_CELL,
} type_kind;
```

`RT_OBJECT` (Section 19.1): This kind of type is not related to any other in any special way. This is the case for `type`, `compiled`, `symbol`, `string`, and `nil`.

`RT_SCALAR` (Chapter 23): The machine scalar types `double`, `int` and `boolean` are of this kind. Values of these types may be unboxed in code specialised to operate on them, or encapsulated in boxes to be treated as reference objects. The `.scal` field points to a structure describing the specific scalar type.

`RT_TYPECONS` (Section 20.3): This kind of type is a "constructor" and, when given arguments in a declaration, can create other types which are "instances". `vector`, `array`, `function` and `cell` are of this kind.

`RT_CALLABLE` (Chapter 21): An object having a type of this kind can be called from user code. The `.sig` field points to a structure describing the argument and return types.

`RT_VECTOR`, `RT_ARRAY` (Chapter 24): Types created by the `vector` and `array` constructors. These refer to the type of elements that their objects can hold via the `.elt` field.

`RT_CELL` (Section 19.4): Types created during the cell introduction phase, of the compiler. Also refers to the type of its contents with `.elt`.

20.1.1 Operations

⟨*types*⟩≡
⟨*r_typetab*⟩
⟨*rtype_equal*⟩
⟨*rtype_hash*⟩
⟨*rtype_gc*⟩
⟨*free_type_data*⟩
⟨*rtype_free*⟩
⟨*elt_print*⟩
⟨*rtype_init_name*⟩
⟨*rtype_print*⟩
⟨*type_ops*⟩

Two types are equal if they're of the same kind, and their kind-specific fields are equal. For types of kind `SCALAR` or `TYPECONS`, pointer equality is sufficient as their descriptors are static data. `OBJECT` types with different names are distinct (these are always static strings, so pointer equality suffices.)

⟨rtype_equal⟩≡

```
bool sig_equal(funsig_t *a, funsig_t *b);
static bool rtype_equal(const void *xp, const void *yp)
{
    const rtype_t *x = xp, *y = yp;

    if(x->kind != y->kind)
        return false;
    switch(x->kind)
    {
    case RT_CALLABLE:
        return sig_equal(x->sig, y->sig);
    case RT_VECTOR:
    case RT_ARRAY:
    case RT_CELL:
        return r_equal(x->elt, y->elt);
    case RT_SCALAR:
        return x->scal == y->scal;
    case RT_TYPECONS:
        return x->cons == y->cons;
    case RT_OBJECT:
        return x->name == y->name;
    }
    return false;
}
```

`.hash` is a simple accessor, as it is for symbols.

⟨rtype_hash⟩≡

```
static uint32_t rtype_hash(const void *ptr)
{
    return ((rtype_t *)ptr)->hash;
}
```

The `.elt` field of a `VECTOR`, `ARRAY` or `CELL` type points to a managed object, which must be marked during garbage collection. The `.sig` field of a `CALLABLE` isn't itself an object, but contains pointers to them, which are marked by `sig_gc`.

⟨rtype_gc⟩≡

```
void sig_gc(funsig_t *sig);
static void rtype_gc(void *ptr)
{
    rtype_t *type = ptr;
    assert(r_typeof(type) == r_type_type);
    switch(type->kind)
    {
    case RT_CALLABLE:
        sig_gc(type->sig);
        break;
    case RT_VECTOR:
    case RT_ARRAY:
    case RT_CELL:
        gc_mark(type->elt);
        break;
    default:
```

```
            break;
        }
    }
```

The name of a `VECTOR`, `ARRAY`, or `CELL` type has the form “`label(element)`”, where `element` is the name of the type’s `.elt` field.

⟨*elt_print*⟩≡

```
static inline void elt_print(FILE *fp, const char *label, const rtype_t *type)
{
    fprintf(fp, "%s(", label);
    r_print(fp, type->elt);
    fprintf(fp, ")");
}
```

A `CALLABLE` type has a name of the form “`function(args):ret`”, and is produced by `sig_print`.

These kinds of type have lazily initialised `.name` fields; they start as `NULL`, and are computed when required.

⟨*rtype_init_name*⟩≡

```
void sig_print(FILE *fp, funsig_t *sig);
char *rtype_init_name(const rtype_t *type)
{
    char *buf = NULL;
    size_t sz;
    FILE *fp = open_memstream(&buf, &sz);
    assert(fp);
    switch(type->kind)
    {
    case RT_CALLABLE:
        sig_print(fp, type->sig);
        break;
    case RT_VECTOR:
        elt_print(fp, "vector", type);
        break;
    case RT_ARRAY:
        elt_print(fp, "array", type);
        break;
    case RT_CELL:
        elt_print(fp, "cell", type);
        break;
    default:
        break;
    }
    fclose(fp);
    return buf;
}
```

The `rtype_name` accessor will initialise a type’s `.name` field on demand.

⟨*rtype_name*⟩≡

```
static inline const char *rtype_name(rtype_t *type)
{
    if(!type->name)
        type->name = rtype_init_name(type);
    return type->name;
}
```

This is invoked when a printed representation of the type is required, such as in the `ops.print` callback, or compiler diagnostic message output.

⟨*rtype_print*⟩≡
```
static void rtype_print(FILE *fp, const void *ptr)
{
    fprintf(fp, "%s", rtype_name((rtype_t *)ptr));
}
```

Types are immutable and unique, so are also internalised in the manner of symbols. The type table `r_typetab` contains all currently live types.

⟨*r_typetab*⟩≡
```
hashset_t *r_typetab;
```

When a type is collected as garbage, it can be removed from the table.

⟨*rtype_free*⟩≡
```
static void rtype_free(void *ptr)
{
    hashset_remove(r_typetab, ptr);
    free_type_data(ptr);
}
```

Lazily initialised `.name`s must be deallocated on destruction. A `CALLABLE` type also needs its `funsig_t` freed.

⟨*free_type_data*⟩≡
```
void sig_free(funsig_t *sig);
static void free_type_data(rtype_t *type)
{
    switch(type->kind)
    {
    case RT_CALLABLE:
        sig_free(type->sig);
        /* fallthrough */
    case RT_VECTOR:
    case RT_ARRAY:
    case RT_CELL:
        if(type->name)
            xfree(type->name);
        break;
    default:
        break;
    }
}
```

⟨*type_ops*⟩≡
```
static const typeops_t type_ops = {
    .free = rtype_free,
    .gc = rtype_gc,
    .hash = rtype_hash,
    .equal = rtype_equal,
    .print = rtype_print
};
```

20.1.2 Predicates

Various builtin functions or compiler optimisations might only apply to objects having a certain kind of type. Each simple predicate function returns `true` when its argument satisfies the test.

⟨*simple predicates*⟩≡
```
static inline bool rtype_is_vector(const rtype_t *typ)
    { return typ->kind == RT_VECTOR; }
static inline bool rtype_is_array(const rtype_t *typ)
    { return typ->kind == RT_ARRAY; }
static inline bool rtype_is_scalar(const rtype_t *typ)
    { return typ->kind == RT_SCALAR; }
static inline bool rtype_is_callable(const rtype_t *typ)
    { return typ->kind == RT_CALLABLE; }
static inline bool rtype_is_constructor(const rtype_t *typ)
    { return typ->kind == RT_TYPECONS; }
static inline bool rtype_is_cell(const rtype_t *typ)
    { return typ->kind == RT_CELL; }
```

20.1.3 Container Types

A *container* type is either a `VECTOR` or an `ARRAY` (Chapter 24).

⟨*simple predicates*⟩+≡
```
static inline bool rtype_is_container(const rtype_t *typ)
    { return typ->kind == RT_VECTOR || typ->kind == RT_ARRAY; }
```

`rtype_eltsz` returns the size of an object of the given type, in bytes, when stored as an element in a container (since unboxed values of `SCALAR` types can differ in size.)

⟨*rtype_eltsz*⟩≡
```
static inline size_t rscal_size(const rtype_t *typ);
static inline size_t rtype_eltsz(const rtype_t *typ)
{
    assert(typ);
    return rtype_is_scalar(typ)
        ? rscal_size(typ)
        : sizeof(robject_t *);
}
```

20.2 Creation

⟨*creation*⟩≡
⟨*compute_hash*⟩
⟨*type_init*⟩
⟨*rtype_create*⟩
⟨*r_global_types*⟩
⟨*rtype_install*⟩
⟨*rtype_init*⟩
⟨*rtype_cons_init*⟩

A type is initialised by populating its fields and computing its hash. `ops` may be `NULL` if the type won't ever be referred to by the `.type` field of any object – this is true of type constructors. `name` may be `NULL` if it will be lazily initialised. Assignment to the `.data` field initialises the union of pointers in a kind-oblivious manner (albeit one not strictly standards-conforming.)

⟨*type_init*⟩≡
```
static void type_init(rtype_t *type, type_kind kind, void *data,
                      const typeops_t *ops, const char *name)
{
    *type = (rtype_t) {
        .base.type = r_type_type,
        .name = (char *)name,
        .kind = kind,
        .data = data,
        .ops = ops,
        .hash = 0,
    };
    type->hash = compute_hash(type);
}
```

A `CALLABLE` type takes the hash of its signature. Other kinds compute the hash of the type object itself.

⟨*compute_hash*⟩≡
```
static uint32_t compute_hash(rtype_t *type)
{
    if(type->kind == RT_CALLABLE)
        return type->sig->hash;
    return hash_code(type, sizeof(rtype_t));
}
```

A temporary `rtype_t` is populated, hashed, and used to probe the type table. If found, the equivalent object is returned, else a new one is allocated with the contents of the temporary.

⟨*rtype_create*⟩≡
```
rtype_t *rtype_create(type_kind kind, void *data,
                      const typeops_t *ops, const char *name)
{
    rtype_t *type, *otyp, tmp;

    type_init(&tmp, kind, data, ops, name);
    otyp = hashset_get(r_typetab, &tmp);
    if(!otyp)
    {
        type = gc_alloc(r_type_type, sizeof(rtype_t));
        *type = tmp;
        hashset_insert(r_typetab, type);
        return type;
    }
    free_type_data(&tmp);
    return otyp;
}
```

The global type environment maps names to types.

⟨*r_global_types*⟩≡
```
hashmap_t *r_global_types;
```

A type may be installed there to make it available for the user to mention by name in type specifiers.

⟨*rtype_install*⟩≡
```
void rtype_install(rtype_t *type, const char *name)
{
    hashmap_set(r_global_types, r_intern(name), type);
}
```

`rtype_init` creates and installs a type, returning it for assignment to one of the `r_type_` variables.

⟨*rtype_init*⟩≡
```
rtype_t *rtype_init(type_kind kind, const typeops_t *ops, const char *name)
{
    rtype_t *type = rtype_create(kind, NULL, ops, name);
    rtype_install(type, name);
    return type;
}
```

20.3 Type Constructors

Types of kind `TYPECONS` can construct new types, given types as arguments.

⟨*constructors*⟩≡
  ⟨*rtype_cons_create*⟩
  ⟨*rtype_from_name*⟩
  ⟨*rtype_from_node*⟩
  ⟨*rtype_from_spec*⟩

Each type constructor is described by a `consdesc_t`. `.from_spec` is invoked when this type constructor is found in a type specifier, and is responsible for creating a new instance. The kind of type which it creates is recorded in the `.kind` field.

⟨*consdesc_t*⟩≡
```
typedef struct ast ast_t;
typedef struct consdesc consdesc_t;
typedef rtype_t *(*typecons_fn)(const consdesc_t *, unsigned, ast_t *);
typedef struct consdesc
{
    typecons_fn from_spec;
    type_kind kind;
} consdesc_t;
```

Testing whether a type is an instance of a given type constructor is a matter of matching the instance's `.kind` with the constructor's `.cons.kind`.

⟨*simple predicates*⟩+≡
```
static inline bool
rtype_is_instance(const rtype_t *self, const rtype_t *other)
    { return other->kind == RT_TYPECONS && other->cons
          && other->cons->kind == self->kind; }
```

Convenience functions allow the creation of type constructors…

⟨*rtype_cons_create*⟩≡
```
rtype_t *rtype_cons_create(const consdesc_t *cons, const char *name)
{
    return rtype_create(RT_TYPECONS, (void *)cons, NULL, name);
}
```

. . . and their subsequent installation in the global type environment.

⟨*rtype_cons_init*⟩≡
```
rtype_t *rtype_cons_init(const consdesc_t *cons, const char *name)
{
    rtype_t *type = rtype_cons_create(cons, name);
    rtype_install(type, name);
    return type;
}
```

20.4 Type Specifiers

A declaration or `type` expression in user code specifies a type – either with a type name, or the application of a type constructor (to other type specifiers.)

Given an `ast_t` representing a type specifier, `rtype_from_spec` finds or creates the corresponding `rtype_t`. `NULL` will be returned if this could not be done – this may or may not be an error, depending on the caller.

The `INVALID` AST is present when the return type of a function has been omitted. A `NAME` or a `SYMBOL` is a simple specifier, as is a `TOKEN`. A `NODE` is a compound specifier denoting a type instance.

Other ASTs are not expected in type specifiers.

⟨*rtype_from_spec*⟩≡
```
rtype_t *rtype_from_spec(ast_t *ast)
{
    if(!ast)
        return NULL;
    switch(ast->type)
    {
    case AST_INVALID:
        return NULL;
    case AST_NAME:
    case AST_SYMBOL:
        return rtype_from_name(ast->symbol);
    case AST_TOKEN:
        return rtype_from_name(r_intern(ast_str(ast)));
    case AST_NODE:
        assert(alen(ast->children) >= 1);
        return rtype_from_node(ast->children);
    default:
        break;
    }
    c_warning("malformed type declaration.");
    return NULL;
}
```

Names are looked up in the global type environment. Since this resolution is done at compile-time, a warning can be emitted if no type by that name could be found.

⟨*rtype_from_name*⟩≡
```
static rtype_t *rtype_from_name(rsymbol_t *name)
{
    rtype_t *type = hashmap_get(r_global_types, name, NULL);

    if(!type)
        c_warning("unknown type ‘%s‘.", r_symstr(name));
    return type;
}
```

The first element in the AST node should specify a type constructor; if it doesn't, a warning is issued and `NULL` is returned.

Its `cons.from_spec` callback is invoked, passing `.cons`, the argument count, and the remainder of the `ast_t` array (usually the elements therein denote type specifiers, so it'll call `rtype_from_spec` recursively.)

⟨*rtype_from_node*⟩≡
```
static rtype_t *rtype_from_node(ast_array_t *arr)
{
    int nargs = alen(arr)-1;
    ast_t *car = aptr(arr, 0);
    ast_t *args = (nargs>0) ? aptr(arr,1) : NULL;
    rtype_t *type = rtype_from_spec(car);

    if(!type)
        return NULL;
    if(!rtype_is_constructor(type))
    {
        c_warning("type ‘%s‘ not a type constructor.", ast_str(car));
        return NULL;
    }
    assert(type->cons && type->cons->from_spec);
    return type->cons->from_spec(type->cons, nargs, args);
}
```

20.5 Subtyping

When `x` is a subtype of `y`, an object of type `x` can be used in a place where one of type `y` is expected – as the right-hand side of an assignment or as a function argument, for example.

⟨*subtyping*⟩≡
  ⟨*r_type_compat*⟩
  ⟨*r_common_type*⟩

In object-oriented languages, subtyping usually follows inheritance. Here, we have a very simple subtyping relation, existing mostly to simplify generic code (such as functions that can operate on vectors of any element type). The subtype test is useful for making decisions at run-time.

The compiler has the slightly more detailed notion of *compatibility*. Although it has the same semantics as subtyping, when the test for compatibility is carried out during compilation it can have one of three results – true, false, or "maybe". In the last case, it could be true for some run-time objects and false for others. A check or conversion must be inserted at that point to ensure correctness.

⟨*compat*⟩≡
```
typedef enum { NO, YES, MAYBE } compat;
```

`r_type_compat` tests whether a `self` is compatible with a place expecting an `other`. `unbox` is `true` if scalars can be treated as unboxed values; `false` if scalars should, like all other values, be considered reference objects.

⟨*r_type_compat*⟩≡
```
bool sig_types_equal(funsig_t *self, funsig_t *other);
compat r_type_compat(const rtype_t *self, const rtype_t *other, bool unbox)
{
    assert(self);
    assert(other);
    ⟨decide compatibility⟩
}
```

When this is the case, compatibility is equivalent to subtyping – `int` is not a subtype of `double`.

⟨*r_subtypep*⟩≡
```
static inline bool r_subtypep(const rtype_t *self, const rtype_t *other)
    { return r_type_compat(self, other, false) == YES; }
```

Compatibility is reflexive – a type is compatible with itself.

⟨*decide compatibility*⟩≡
```
if(self == other)
    return YES;
```

When scalar values are considered (i.e. at compile-time) implicit arithmetic conversion is possible.

⟨*decide compatibility*⟩+≡
```
if(unbox && rtype_is_scalar(self) && rtype_is_scalar(other))
    return MAYBE;
```

A value of statically unknown type (i.e. `object`) will have a more specific type at run-time, so must be checked.

⟨*decide compatibility*⟩+≡
```
if(self == r_type_object)
    return MAYBE;
```

A place expecting an `object` can handle objects of any type, but scalars must be boxed.

⟨*decide compatibility*⟩+≡
```
if(other == r_type_object)
    return (unbox & rtype_is_scalar(self)) ? MAYBE : YES;
```

`nil` is compatible with every reference type.

⟨*decide compatibility*⟩+≡
```
if(self == r_type_nil)
    return rtype_is_scalar(other) ? NO : YES;
```

A value of any reference type could be `nil`, so must be checked at run-time.

⟨*decide compatibility*⟩+≡
```
if(other == r_type_nil)
    return rtype_is_scalar(self) ? NO : MAYBE;
```

Every instance of a type is compatible with its constructor – `vector(int)` is compatible with `vector`, for example.

⟨*decide compatibility*⟩+≡
```
if(rtype_is_instance(self, other))
    return YES;
```

A value that's compatible with a constructor might also be compatible with a specific instance – if all the compiler knows about a value is that it's a `vector`, for example, it will have to insert a check before a site which expects a `vector(double)`.

⟨*decide compatibility*⟩+≡
```
if(rtype_is_instance(other, self))
    return MAYBE;
```

Callables (that is, user-visible functions) are compatible when they have identical argument and return types (names are ignored.)

⟨*decide compatibility*⟩+≡
```
if(self->kind == RT_CALLABLE && other->kind == RT_CALLABLE)
    return sig_types_equal(self->sig, other->sig) ? YES : NO;
```

No other types are compatible.

⟨*decide compatibility*⟩+≡
```
return NO;
```

Given two types (for example, those of the consequent and alternate branches of an `if`) the compiler may need to find a type which is a supertype of both (the type of the `if` expression.)

For precision's sake, this type should be the most specific possible. If either type is a subtype of the other, then the latter is chosen. If they're scalars, picking the wider of the two enables arithmetic conversion. If they share a common type constructor, it's chosen. Otherwise the only possibility is `object`.

⟨*r_common_type*⟩≡
```
rtype_t *r_common_type(rtype_t *self, rtype_t *other)
{
    assert(self);
    assert(other);
    if(r_subtypep(self, other))
        return other;
    if(r_subtypep(other, self))
        return self;
    if(self->kind != other->kind)
        return r_type_object;
    switch(self->kind)
    {
    case RT_SCALAR: return rscal_promote(self, other);
    case RT_CALLABLE: return r_type_callable;
    case RT_VECTOR: return r_type_vector;
    case RT_ARRAY: return r_type_array;
    default: break;
    }
    return r_type_object;
}
```

20.6 Initialisation

The global type environment and type tables are created at initialisation time. The latter contains only types, so the specialised `rtype_hash` and `rtype_equal` functions can be used.

Unusually for an object, `r_type_type`'s `.type` is `r_type_type`. This otherwise illegal self-reference is facilitated via collusion with the allocator (Section 25.4). The `.type` field can then be correctly initialised.

⟨*rt_bootstrap*⟩≡
```
rtype_t *r_type_type;
void rt_bootstrap()
{
    r_global_types = hashmap_create(rsym_hash, ptr_eq);
    r_typetab = hashset_create(rtype_hash, rtype_equal);
    r_type_type = rtype_create(RT_OBJECT, NULL, &type_ops, "type");
    r_type_type->base.type = r_type_type;
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "ast.h"
```

⟨*rt/type.h*⟩≡
```
⟨type_kind⟩
⟨consdesc_t⟩
⟨rtype_t⟩
⟨compat⟩
⟨prototypes⟩
⟨externs⟩

⟨simple predicates⟩
⟨rtype_eltsz⟩
⟨r_subtypep⟩
⟨rtype_name⟩
```

⟨*externs*⟩≡
```
extern hashmap_t *r_global_types;
extern hashset_t *r_typetab;
extern rtype_t *r_type_type;
```

⟨*prototypes*⟩≡
```
compat r_type_compat(const rtype_t *self, const rtype_t *other, bool unbox);
rtype_t *r_common_type(rtype_t *self, rtype_t *other);
char *rtype_init_name(const rtype_t *type);
rtype_t *rtype_create(type_kind kind, void *data,
                      const typeops_t *ops, const char *name);
rtype_t *rtype_cons_create(const consdesc_t *cons, const char *name);
rtype_t *rtype_init(type_kind kind, const typeops_t *ops, const char *name);
rtype_t *rtype_cons_init(const consdesc_t *cons, const char *name);
void rtype_install(rtype_t *type, const char *name);
rtype_t *rtype_from_spec(ast_t *ast);
void rt_bootstrap();
```

Chapter 21

Functions, Closures, Builtins

A *callable* object can be invoked by a function call expression. They participate in the calling convention established by the VM (Chapter 22).

⟨*rt/callable.c*⟩≡
  ⟨*includes*⟩
  ⟨*callables*⟩
  ⟨*signatures*⟩
  ⟨*type construction*⟩
  ⟨*closures*⟩
  ⟨*functions*⟩
  ⟨*builtins*⟩
  ⟨*rt_install_call_types*⟩

To the user, callable objects have types which are instances of `function`. This type constructor is known as `r_type_callable` to the runtime.

21.1 Callables

A callable is either a *closure* or a *builtin*. Function expressions create closures, while builtins allow C functions to be invoked by the VM.

⟨*call_dispatch*⟩≡
```
typedef enum { CALL_CLOSURE, CALL_BUILTIN } call_dispatch;
```

Both begin with a common `rcallable_t` structure.

⟨*rcallable_t*⟩≡
```
typedef struct rcallable
{
    robject_t base;
    call_dispatch disp;
} rcallable_t;
```

They can be distinguished by `rcall_is_builtin`.

⟨*rcall_is_builtin*⟩≡
```
static inline bool rcall_is_builtin(const rcallable_t *cl)
    { return cl->disp != CALL_CLOSURE; }
```

21.1.1 Operations

⟨*callables*⟩≡
  ⟨*rcall_gc*⟩
  ⟨*rcall_free*⟩
  ⟨*rcall_print*⟩
  ⟨*call_ops*⟩

A closure may allocate an array containing references to managed objects, so must be handled appropriately by the `ops.gc`...

⟨*rcall_gc*⟩≡
```
static void closure_gc(rclosure_t *cl);
static void rcall_gc(void *ptr)
{
    if(!rcall_is_builtin(ptr))
        closure_gc(ptr);
}
```

...and `ops.free` callbacks.

⟨*rcall_free*⟩≡
```
static void closure_free(rclosure_t *cl);
static void rcall_free(void *ptr)
{
    if(!rcall_is_builtin(ptr))
        closure_free(ptr);
}
```

A callable has the printed representation "`<type pointer>`". Some builtins have more information available; their `.name` may be printed instead.

⟨*rcall_print*⟩≡
```
static void rcall_print(FILE *fp, const void *ptr)
{
    const rcallable_t *cl = ptr;

    fprintf(fp, "<");
    r_print(fp, r_typeof(ptr));
    if(rcall_is_builtin(cl))
    {
        const cbuiltin_t *cbi = ((rbuiltin_t *)ptr)->cbi;

        if(cbi)
        {
            fprintf(fp, " `%s`>", cbi->name);
            return;
        }
    }
    fprintf(fp, " %p>", ptr);
}
```

Two callables are equal only when they are the same object, so the `ops.equal` callback is omitted.

⟨*call_ops*⟩≡
```
static const typeops_t call_ops = {
    .free = rcall_free,
    .gc = rcall_gc,
    .print = rcall_print
};
```

21.2 Signatures

An `rtype_t` with `.kind` `CALLABLE` is called a *signature*.

⟨*signatures*⟩≡
⟨*sig_equal*⟩
⟨*sig_types_equal*⟩
⟨*sig_print*⟩
⟨*argdesc_mark*⟩
⟨*sig_gc*⟩
⟨*sig_free*⟩

The type's `.sig` field points to an auxiliary `funsig_t` structure that it owns.

`.args` is an array of length `.nargs`, each element describing the corresponding argument. `.ret_type` is the type of the return value. `.has_rest` is set if `...` is present as an argument. `reqbits` has one bit per argument, the inverse of `.is_optional`.

During a call, argument values are placed on the stack in a "call frame" (Section 26.1). `.argsz` is its total size, in bytes.

⟨*funsig_t*⟩≡
```
typedef struct funsig
{
    argdesc_t *args;
    unsigned nargs;
    rtype_t *ret_type;
    bool has_rest;
    unsigned reqbits;
    uint32_t hash;
    op_offset_t argsz;
} funsig_t;
```

When a function with this signature is called, each argument must be a subtypes of its corresponding `.type`. Its value is stored `.offset` bytes from the start of the call frame. `.name` may be `NULL`; if not, it can be named in a call expression. `.is_optional` is true when the argument has a default expression, and so is not required to be present in a call.

⟨*argdesc_t*⟩≡
```
typedef struct
{
    rtype_t *type;
    op_offset_t offset;
    rsymbol_t *name;
    bool is_optional;
} argdesc_t;
```

Two `CALLABLE` types with the same arguments and return types are equal (since the other values are derived.) Types are tested with pointer equality, as they've been interned by construction.

⟨*sig_equal*⟩≡
```
bool sig_equal(funsig_t *a, funsig_t *b)
{
    if(a->nargs != b->nargs || a->ret_type != b->ret_type)
        return false;
    for(int i=0; i<a->nargs; i++)
    {
        argdesc_t *m = &a->args[i], *n = &b->args[i];

        if(m->name != n->name ||
           m->is_optional != n->is_optional ||
           m->type != n->type ||
           m->offset != n->offset)
            return false;
    }
    return true;
}
```

For subtyping and compatibility, only argument and return types need to be checked, as type declarations don't include argument names or optional flags.

⟨*sig_types_equal*⟩≡
```
bool sig_types_equal(funsig_t *a, funsig_t *b)
{
    if(a->nargs != b->nargs || a->ret_type != b->ret_type)
        return false;
    for(int i=0; i<a->nargs; i++)
    {
        argdesc_t *m = &a->args[i], *n = &b->args[i];

        if(m->type != n->type)
            return false;
    }
    return true;
}
```

A signature prints like its type specifier, "`function (args) : return`". `...`, if present, has a known type and is always optional, so those need not be printed. A trailing "`=`" on an argument denotes its optionality. If the return type is `object`, it is omitted.

⟨*sig_print*⟩≡
```
void sig_print(FILE *fp, funsig_t *sig)
{
    fprintf(fp, "function ");
    fputc('(', fp);
    for(int i=0; i<sig->nargs; i++)
    {
        argdesc_t *arg = &sig->args[i];

        if(i > 0)
            fprintf(fp, ", ");
        if(arg->name == r_sym_rest)
            r_print(fp, arg->name);
        else if(arg->name)
        {
            if(arg->type != r_type_object)
```

```
            {
                r_print(fp, arg->type);
                fputc(' ', fp);
            }
            r_print(fp, arg->name);
            if(arg->is_optional)
                fputc('=', fp);
        }
        else
            r_print(fp, arg->type);
    }
    fputc(')', fp);
    if(sig->ret_type != r_type_object)
    {
        fprintf(fp, " : ");
        r_print(fp, sig->ret_type);
    }
}
```

Types and symbols are managed objects, so a signature must participate in garbage collection by marking them…

⟨*sig_gc*⟩≡
```
void sig_gc(funsig_t *sig)
{
    gc_mark(sig->ret_type);
    for(int i=0; i<sig->nargs; i++)
        argdesc_mark(&sig->args[i]);
}
```

…in each argument descriptor.

⟨*argdesc_mark*⟩≡
```
static inline void argdesc_mark(argdesc_t *arg)
{
    gc_mark(arg->name);
    gc_mark(arg->type);
}
```

The type will free its `funsig_t` during deallocation.

⟨*sig_free*⟩≡
```
void sig_free(funsig_t *sig)
{
    if(sig->args)
        xfree(sig->args);
    xfree(sig);
}
```

And given the former, the latter may be extracted with `rcall_sig`.

⟨*rcall_sig*⟩≡
```
static inline funsig_t *rcall_sig(const rcallable_t *cl)
    { return cl->base.type->sig; }
```

21.3 Type Construction

`CALLABLE` types can be created by type expressions in user code, or by the compiler and runtime as needed (for example, when installing new builtins.)

⟨*type construction*⟩≡
⟨*copy_args*⟩
⟨*rcall_type_create*⟩
⟨*call_type_from_spec*⟩
⟨*callable_cons*⟩

The latter case is handled by `rcall_type_create`, which must be provided a return type, an argument count, and an array of `argdesc_t`s with their `.name`, `.type` and `.is_optional` fields initialised as desired.

(Note that `.argsz` always includes at least an `argbits_t`. This could be omitted for functions without optional arguments, at the cost of complicating function invocation.)

`.hash` is computed from the hashes of the return type and the argument list – `.reqbits`, `.nargs`, `.argsz` and `.has_rest` are derived from the latter and so need not be included.

⟨*rcall_type_create*⟩≡
```
rtype_t *rcall_type_create(rtype_t *ret_type, unsigned nargs,
                           argdesc_t *args)
{
    funsig_t *sig = xmalloc(sizeof(funsig_t));

    *sig = (funsig_t) {
        .args = nargs > 0 ? xcalloc(nargs, sizeof(argdesc_t)) : NULL,
        .nargs = nargs,
        .reqbits = 0,
        .argsz = sizeof(argbits_t),
        .ret_type = ret_type,
        .has_rest = false
    };
    sig->hash = hash_roll(r_hash(ret_type), copy_args(sig, nargs, args));
    return rtype_create(RT_CALLABLE, sig, &call_ops, NULL);
}
```

The `argdesc_t` array is copied, computing each argument's `.offset`, updating the signature's `.argsz`, `.reqbits` and `.has_rest` fields accordingly.

The hash of the argument list is the hash of each argument's fields as provided by the caller.

⟨*copy_args*⟩≡
```
static uint32_t copy_args(funsig_t *sig, unsigned nargs, argdesc_t *args)
{
    uint32_t hash = 0;

    for(int i=0; i<nargs; i++)
    {
        argdesc_t *arg = &sig->args[i];

        *arg = args[i];
        arg->offset = sig->argsz;
        if(arg->name == r_sym_rest)
            sig->has_rest = true;
        if(!arg->is_optional)
            sig->reqbits |= 1<<i;
        sig->argsz += rtype_eltsz(arg->type);
```

```
        hash = hash_code_seed(&arg->is_optional, sizeof(bool), hash);
        hash = hash_roll(hash, r_hash(arg->name));
        hash = hash_roll(hash, r_hash(arg->type));
    }
    return hash;
}
```

A trivial convenience function exists to initialise them.

⟨*argdesc_init*⟩≡

```
static inline argdesc_t argdesc_init(rsymbol_t *name, rtype_t *type, bool is_opt)
{
    return (argdesc_t) {
        .name = name,
        .type = type,
        .is_optional = is_opt
    };
}
```

When a `function` type constructor is encountered in a type specifier, `call_type_from_spec` is invoked, with an array containing specifiers for the argument and return types.

⟨*callable_cons*⟩≡

```
static consdesc_t callable_cons = {
    .from_spec = call_type_from_spec,
    .kind = RT_CALLABLE
};
```

If none were supplied, the `function` type itself is returned – since this is a supertype of all `CALLABLE`s, it lets the user specify any type of callable object.

Otherwise, the grammar has arranged that the return type is the first specifier in the array. If it was omitted, `object` is assumed.

Each `argdesc_t` takes its `.type` from the corresponding type specifier. The `...` argument, if present, is a named vector of `object`s, optional, and must be the last element in the array.

⟨*call_type_from_spec*⟩≡

```
static rtype_t *
call_type_from_spec(const consdesc_t *cons, unsigned nargs, ast_t *args)
{
    if(nargs < 1)
        return r_type_callable;

    argdesc_t *descs = alloca(sizeof(argdesc_t) * (nargs-1));
    rtype_t *ret_type = (args[0].type == AST_INVALID)
        ? r_type_object
        : rtype_from_spec(&args[0]);

    if(!ret_type)
        return NULL;
    for(int i=1; i<nargs; i++)
    {
        argdesc_t *desc = &descs[i-1];
        ast_t *arg = &args[i];

        if(arg->type == AST_NAME && arg->symbol == r_sym_rest)
        {
            if(i != nargs-1)
            {
                c_warning("‘...‘ not last argument in list.");
```

```
                return NULL;
            }
            *desc = argdesc_init(r_sym_rest, (rtype_t *)r_type_vec_object, true);
        }
        else
        {
            rtype_t *type = rtype_from_spec(arg);

            if(!type)
                return NULL;
            *desc = argdesc_init(NULL, type, false);
        }
    }
    return rcall_type_create(ret_type, nargs-1, descs);
}
```

21.4 Closures

Function expressions in user code cause closures to be created. The closure provides access to the values of variables bound by lexically enclosing functions.

⟨*closures*⟩≡
  ⟨*closure_free*⟩
  ⟨*closure_gc*⟩
  ⟨*rcall_closure_create*⟩

This object of `CALLABLE` type is a pair of pointers. `.fn` is the function compiled from its corresponding expression; `.env` a buffer containing values saved from its lexical environment. This corresponds to the "flat closure" arrangement described in Serrano and Weis (1995).

⟨*rclosure_t*⟩≡
```
typedef struct rclosure
{
    rcallable_t base;
    rfunction_t *fn;
    void *env;
} rclosure_t;
```

The `.disp` of a closure is always `CALL_CLOSURE`. If the function needs no environment, `.env` is left `NULL` (it would be possible to avoid creating a closure entirely for functions which don't capture lexical variables, at the cost of further complicating function invocation).

⟨*rcall_closure_create*⟩≡
```
rclosure_t *rcall_closure_create(rtype_t *type, rfunction_t *fn)
{
    rclosure_t *cl = gc_alloc(type, sizeof(rclosure_t));
    op_offset_t env_sz = fn->env_sz;

    cl->base.disp = CALL_CLOSURE;
    cl->fn = fn;
    cl->env = (env_sz > 0) ? xcalloc(1, env_sz) : NULL;
    return cl;
}
```

The compiled function is a managed object – it's shared by all closures over a given function expression – and so must be marked during garbage collection.

The environment is split into two areas - scalars are stored from bytes 0 to `.env_scalsz - 1`, and references from `.env_scalsz` to `.env_sz - 1`. Each pointer in the latter must be marked.

⟨*closure_gc*⟩≡

```
#define env_ptr(c, f) (robject_t **)((uint8_t *)c->env + c->fn->f)
static void closure_gc(rclosure_t *cl)
{
    gc_mark(cl->fn);
    if(cl->env)
    {
        robject_t **ptr;

        for(ptr = env_ptr(cl, env_scalsz);
            ptr < env_ptr(cl, env_sz); ptr++)
            gc_mark(*ptr);
    }
}
```

As `.env` was allocated, it must be freed.

⟨*closure_free*⟩≡

```
static void closure_free(rclosure_t *cl)
{
    if(cl->env)
        xfree(cl->env);
}
```

21.5 Functions

Internal to the runtime subsystem, a *function* is the compiled representation of a single function expression in user code. It is not, in itself, callable.

⟨*functions*⟩≡

```
⟨rfunc_create⟩
⟨rfunc_free⟩
⟨rfunc_gc⟩
⟨func_ops⟩
```

The VM interprets a stream of `op_code_t` opcodes as instructions. Each instruction is typically followed by a number of `op_stack_t` or `op_offset_t` operands.

(The former is usually a byte displacement from the stack frame base pointer – negative values to address function arguments, positive for local variables. The latter, a zero-based byte offset or element index into some other structure.)

The `argbits_t` typedef is merely informative. Argument bitflags are manipulated as `rint_t`s by the VM, so these types must be the same size.

⟨*bytecode types*⟩≡

```
typedef uint8_t op_code_t;
typedef uint16_t op_offset_t;
typedef int16_t op_stack_t; /* "32k of stack frame is enough for anyone" */
typedef unsigned argbits_t;
static_assert(sizeof(argbits_t) == sizeof(rint_t),
              "argbits not the same size as int.");
```

When the argument at position `b` was omitted from a call with argument bitflags `a`, `argbits_missing` returns `true`.

⟨*argbits_missing*⟩≡
```
static inline bool argbits_missing(argbits_t a, unsigned b)
    { return (a & (1<<b)) == 0; }
```

Each user function generates at most one `rfunction_t` (some may be removed by dead code elimination.)

The function can be referred to by many closures, so is a managed object. The body of the function is compiled into the array stored in the `.code` field. The `.cl_type` field is the signature of the function (and so the `.type` of its closures.)

Local variable storage (`.loc_scalsz`, `.loc_sz`) and the closure environment (`.env_scalsz`, `.env_sz`) are both split into areas where values of scalar and reference types are stored. Neither are mapped in any more detail, as the exact layouts are only needed during code generation.

`.consts` holds an array of `nconsts` references which serve as the function's "constant pool". `CONST` opcodes refer to objects from here.

⟨*rfunction_t*⟩≡
```
typedef struct
{
    robject_t base;
    op_code_t *code;
    rtype_t *cl_type;
    op_offset_t loc_scalsz, loc_sz;
    op_offset_t env_scalsz, env_sz;
    robject_t **consts;
    unsigned nconsts;
} rfunction_t;
```

`rfunc_create` gives the newly-created function ownership of its `code` and the contents of the `consts` array.

⟨*rfunc_create*⟩≡
```
rfunction_t *rfunc_create(rtype_t *cl_type, op_code_t *code, size_t loc_scalsz,
                          size_t loc_sz, size_t env_scalsz, size_t env_sz,
                          robject_array_t *consts)
{
    rfunction_t *rfn = gc_alloc(r_type_function, sizeof(rfunction_t));
    unsigned nconsts = alen(consts);

    *rfn = (rfunction_t) {
        .base = rfn->base,
        .code = code,
        .cl_type = cl_type,
        .loc_scalsz = loc_scalsz,
        .loc_sz = loc_sz,
        .env_scalsz = env_scalsz,
        .env_sz = env_sz,
        .consts = array_cede(consts),
        .nconsts = nconsts
    };
    return rfn;
}
```

Constants in the pool (and the function signature) are managed objects, and must be marked during garbage collection.

⟨rfunc_gc⟩≡
```
static void rfunc_gc(void *ptr)
{
    rfunction_t *fn = ptr;
    gc_mark(fn->cl_type);
    for(int i=0; i<fn->nconsts; i++)
        gc_mark(fn->consts[i]);
}
```

The constant pool and code array are owned by the function, so must be freed when it's collected as garbage.

⟨rfunc_free⟩≡
```
static void rfunc_free(void *ptr)
{
    rfunction_t *fn = ptr;
    if(fn->consts)
        xfree(fn->consts);
    xfree(fn->code);
}
```

Compiled function objects are not user-visible, so don't need the other `.ops` callbacks.

⟨func_ops⟩≡
```
static const typeops_t func_ops = {
    .free = rfunc_free,
    .gc = rfunc_gc,
};
```

21.6 Builtins

The VM can call C functions through the *builtin* mechanism. The compiler can also be informed of their behaviour, and generate bytecode in place of a regular call; optional callbacks specify when and how.

⟨builtins⟩≡
  ⟨rcall_builtin_create⟩
  ⟨count_args⟩
  ⟨argdesc_from_init⟩
  ⟨rbuiltin_define⟩
  ⟨rbuiltin_install⟩

An `rbuiltin_t` is an object of `CALLABLE` type. The `.fn` field points to the C function to invoke. When the compiler comes upon one of these objects, it looks to the `.cbi` field, where present, to determine its behaviour.

⟨rbuiltin_t⟩≡
```
typedef struct rbuiltin rbuiltin_t;
typedef struct vm_ctx vm_ctx_t;
typedef void (*builtin_fn)(vm_ctx_t *, rbuiltin_t *, uint8_t *, void *);
typedef struct rbuiltin
{
    rcallable_t base;
    builtin_fn fn;
    const cbuiltin_t *cbi;
} rbuiltin_t;
```

The `.ops` callbacks in the `cbuiltin_t` structure specify the builtin's semantics, implement any custom code generation it requires, and are described in Section 10.5. Any extra information can be referenced via the `.data` field. The printed representation of the associated `rbuiltin_t` will include the `.name` if it's present.

⟨*cbuiltin_t*⟩≡
```
typedef struct builtin_ops builtin_ops_t;
typedef struct cbuiltin
{
    const builtin_ops_t *ops;
    void *data;
    const char *name;
} cbuiltin_t;
```

Builtins are similar to closures, except with `.disp` equal to `CALL_BUILTIN`, a pointer to a C function instead of VM code, and an optional compiler descriptor instead of an environment.

⟨*rcall_builtin_create*⟩≡
```
rcallable_t *
rcall_builtin_create(rtype_t *type, builtin_fn fn, const cbuiltin_t *cbi)
{
    rbuiltin_t *cl = gc_alloc(type, sizeof(rbuiltin_t));

    cl->base.disp = CALL_BUILTIN;
    cl->fn = fn;
    cl->cbi = cbi;
    return (rcallable_t *)cl;
}
```

A module which implements builtins will `#include "rt/builtin.h"`.

⟨*rt/builtin.h*⟩≡
⟨*builtin_init_arg_t*⟩
⟨*builtin_init_t*⟩
⟨*init macros*⟩
⟨*defbuiltin*⟩
⟨*function macros*⟩
⟨*builtin_return*⟩
⟨*builtin externs*⟩
⟨*missingp*⟩

Each builtin is initialised by a static `builtin_init_t` structure. `.fn` and `.cbi` will be copied to their counterparts in `rbuiltin_t`; `.prtype` holds the address of the global containing the return type of the builtin (it can't simply take a value; types are not constant, they're created during runtime initialisation.)

⟨*builtin_init_t*⟩≡
```
typedef struct
{
    char *name;
    builtin_fn fn;
    rtype_t **prtype;
    const cbuiltin_t *cbi;
    const builtin_init_arg_t *args;
} builtin_init_t;
```

`.args` points to the start of a static array of `builtin_init_arg_t` structures.

⟨*builtin_init_arg_t*⟩≡

```
typedef struct
{
    char *name;
    rtype_t **ptype;
    bool is_optional;
} builtin_init_arg_t;
```

These fields are interned (when present), dereferenced and copied, respectively, to initialise an `argdesc_t`.

⟨*argdesc_from_init*⟩≡

```
static inline argdesc_t argdesc_from_init(const builtin_init_arg_t *arg)
{
    return argdesc_init(arg->name ? r_intern(arg->name) : NULL,
                        *arg->ptype,
                        arg->is_optional);
}
```

The `.args` array is terminated with a sentinel `{ 0 }`, so counting its elements is a matter of finding the first one with no `.ptype`.

⟨*count_args*⟩≡

```
static inline unsigned count_init_args(const builtin_init_t *init)
{
    unsigned i = 0;
    for(; init->args[i].ptype; i++);
    return i;
}
```

`rbuiltin_define` creates and installs a builtin, given a static initialiser. Counting and converting the arguments (and, since this is late enough in the runtime initialisation, dereferencing the pointer to the return type) lets us create the `CALLABLE` type; this, along with the other fields from the structure, is enough to create the builtin object. It's installed in the global environment with the given `.name`.

⟨*rbuiltin_define*⟩≡

```
rcallable_t *rbuiltin_define(const builtin_init_t *init)
{
    unsigned nargs = count_init_args(init);
    argdesc_t *args = alloca(sizeof(argdesc_t) * nargs);

    for(int i=0; i<nargs; i++)
        args[i] = argdesc_from_init(&init->args[i]);

    rtype_t *cl_type = rcall_type_create(*init->prtype, nargs, args);
    rcallable_t *cl = rcall_builtin_create(cl_type, init->fn, init->cbi);

    r_defbuiltin(init->name, cl);
    return cl;
}
```

The convenience function `rbuiltin_install` defines every builtin in an array of initialisers (which is also terminated with `{ 0 }`.)

⟨*rbuiltin_install*⟩≡

```
void rbuiltin_install(const builtin_init_t *init)
{
    for(; init->name; init++)
        rbuiltin_define(init);
}
```

The `.args` field of the builtin initialiser is a pointer, and initialising it with a static array requires an explicit cast. The convenience macro `defbuiltin` helps reduce the clutter (or, when a `cbuiltin_t` is needed, `defcbuiltin`.)

⟨*defbuiltin*⟩≡
```
#define defbuiltin(n, f, prt, a...) \
    { init_builtin(n, f, prt, init_args(a)) }
#define defcbuiltin(n, f, prt, o, d, a...) \
    { init_builtin(n, f, prt, init_args(a)), .cbi = init_cbi(o, n, d) }
```

This provides the fields…

⟨*init macros*⟩≡
```
#define init_builtin(n, f, prt, a...) \
    .name = n, .fn = f, .prtype = prt, .args = a
```

…and the argument list, cast to the correct type (with sentinel value appended.)

⟨*init macros*⟩+≡
```
#define init_args(a...) (builtin_init_arg_t []) { a, { } }
```

Instead of taking a pointer to a named static structure, the `.cbi` field can be initialised inline.

⟨*init macros*⟩+≡
```
#define init_cbi(o, n, d) \
    &(cbuiltin_t) { .ops = o, .name = n, .data = d }
```

⟨*function macros*⟩≡
```
#define def_args struct __attribute__ ((__packed__))
#define end_args(args, n) *n = (void *)(args - sizeof(*n))
```

A builtin is given a pointer to a location where it is to place its return value, and it could be of a pointer or a scalar type. A macro hides some of the verbiage.

⟨*builtin_return*⟩≡
```
#define builtin_return(ret, typ, val) (*((typ *)(ret))) = val
```

Optional arguments to a builtin don't have defaults. Instead, the `argbits_t` from the call frame is made available. This can be passed to `argbits_missing`, along with the zero-based index of the argument in question, to test for its absence.

⟨*missingp*⟩≡
```
#define missingp(a,b) argbits_missing(a,b)
```

21.7 Initialisation

The type of compiled functions is stored in `r_type_function`. It's named "`compiled`" but, as no objects of this type are accessible to the user, should never be seen.

The constructor of `CALLABLE` types is named "`function`", as this lends itself to type specifiers which look similar to the function expressions with which they are compatible.

⟨*rt_install_call_types*⟩≡
```
rtype_t *r_type_function, *r_type_callable;
void rt_install_call_types()
{
    r_type_function = rtype_create(RT_OBJECT, NULL, &func_ops, "compiled");
    r_type_callable = rtype_cons_init(&callable_cons, "function");
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
// required for rbuiltin_define
#include "rt/builtin.h"
// required for from_spec
#include "ast.h"
```

⟨*builtin externs*⟩≡
```
rcallable_t *rbuiltin_define(const builtin_init_t *init);
void rbuiltin_install(const builtin_init_t *init);
```

⟨*rt/callable.h*⟩≡
```
⟨bytecode types⟩
⟨argbits_missing⟩
⟨argdesc_t⟩
⟨funsig_t⟩
⟨rfunction_t⟩
⟨call_dispatch⟩
⟨rcallable_t⟩
⟨rcall_is_builtin⟩
⟨rclosure_t⟩
⟨cbuiltin_t⟩
⟨rbuiltin_t⟩
⟨prototypes⟩
⟨externs⟩
⟨argdesc_init⟩
⟨rcall_sig⟩
```

⟨*externs*⟩≡
```
extern rtype_t *r_type_function, *r_type_callable;
```

⟨*prototypes*⟩≡
```
rtype_t *rcall_type_create(rtype_t *ret_type, unsigned nargs, argdesc_t *args);
rfunction_t *rfunc_create(rtype_t *cl_type, op_code_t *code, size_t loc_scalsz,
                          size_t loc_sz, size_t env_scalsz, size_t env_sz,
                          robject_array_t *consts);
rclosure_t *rcall_closure_create(rtype_t *type, rfunction_t *fn);
rcallable_t *rcall_builtin_create(rtype_t *type, builtin_fn fn,
                                  const cbuiltin_t *builtin);
void rt_install_call_types();
```

Chapter 22

Calling Convention

The compiler and VM agree on a particular treatment of argument and return values, so that callable objects can be invoked from call sites which may be oblivious to their signatures. This is the *calling convention*; every function and call site adheres to it.

22.1 Callee

The ordered list of a function's formal arguments, their names and their types, along with its return type, comprises its signature (Section 21.2).

When called, the code of a function expects an `argbits_t` as a first, hidden, argument; furthest below `bp` in the stack frame (

Arguments are expected in the order, and with the types, specified by the signature's `.args` array. Arguments of scalar type are expected to be unboxed into the stack frame, and may undergo arithmetic promotion (Section 23.1). An omitted argument will have been given its zero value.

On return, the function's code indicates the value of the specified `.ret_type` that the caller should take as its result. When this is a scalar, it's not boxed.

22.2 Caller

The expression that calls a function can provide its actual arguments in a number of ways. Each can be *named*, *omitted*, the *rest* vector "`...`"; or the default, *positional.*

Before the call proceeds, the actual arguments must be *matched* with the formal arguments of the callee.

22.2.1 Fast Call

When the compiler knows the signature of the function being called, it can use the *fast call* convention (the call must also not require allocation of a rest vector, but this is an implementation limitation and could be relaxed without difficulty.)

Arguments are matched at compile-time, and opcodes generated to assign their values directly to the appropriate slots in the stack frame of the callee. Scalars passed in this manner are left unboxed. Since the return type is known, scalar return values don't need boxing either.

22.2.2 Universal Call

The *universal call* convention is used in all other cases.

Actual arguments are supplied by sub-opcodes, with the same names and in the same order as the call expression. Argument matching (and rest-vector creation, when necessary) occurs at run-time. Values are passed as reference objects; scalars must be boxed before being used in such a call. The result of a call is also expected to be a reference object; scalars will be boxed upon return.

22.3 Argument Matching

The argument matching process is defined in its own module, used by both compiler and VM.

⟨*rt/call.c*⟩≡

```
⟨includes⟩
⟨call_sequence⟩
⟨nargs_excluding_rest⟩
⟨get_kwd_arg⟩
⟨get_pos_arg⟩
⟨set_argbit⟩
⟨call_match_kwd⟩
⟨call_match_pos⟩
⟨call_match_omit⟩
⟨no_match_rest⟩
⟨call_match_rest⟩
```

22.3.1 Sequencing

Given the `args` and their `names`, the `call_sequence` function invokes the provided callbacks in a specific sequence, passing along the provided user data `ptr`. All invocations must succeed in order for `true` to be returned.

⟨*call_sequence*⟩≡

```
bool call_sequence(void *ptr, void **args, rsymbol_t **names, int nargs,
                   bool (*rest_fn)(void *, void *),
                   bool (*kwd_fn)(void *, rsymbol_t *, void *),
                   bool (*pos_fn)(void *, void *),
                   bool (*omit_fn)(void *))
{
    bool r = true;

    if(names)
    {
        ⟨named⟩
    }
    ⟨positional⟩
    return r;
}
```

First, `kwd_fn` is called on each argument with a `name` (that isn't "...").

⟨*named*⟩≡

```
for(int i = 0; r && i < nargs; i++)
{
    void *arg = args[i];
    rsymbol_t *name = names[i];

    if(name && name != r_sym_rest)
        r &= kwd_fn(ptr, name, arg);
}
```

Then each remaining argument, in the order given, is passed to one of the other callbacks: "`...`" to `rest_fn`, `NULL` to `omit_fn`, otherwise `pos_fn`.

⟨*positional*⟩≡
```
for(int i = 0; r && i < nargs; i++)
{
    void *arg = args[i];

    if(names && names[i] == r_sym_rest)
        r &= rest_fn(ptr, arg);
    else if(names && names[i])
        continue;
    else if(arg)
        r &= pos_fn(ptr, arg);
    else
        r &= omit_fn(ptr);
}
```

22.3.2 Matchers

The compiler passes the matcher functions to `call_sequence` when optimising a `CALL` node to `CALL_FAST` (Section 13.4).

It also invokes `call_sequence` when generating code for a universal call with a different set of callbacks, emitting a sub-opcode for each argument. The VM then interprets these, invoking the matchers during execution (Subsection 26.4.1).

The `call_ctx_t` structure provides assignment callbacks to the matchers, as well as the signature `.sig` of the function being called. `.posbits` tracks the arguments already assigned to, and so unavailable for use by positionals. `.argbits` tracks the arguments that are present in the call, and will be passed to the callee.

Matchers and their callbacks return `false` on failure.

⟨*call_ctx_t*⟩≡
```
typedef struct
{
    argbits_t posbits; // arguments unavailable for use by positionals
    argbits_t argbits; // arguments present in call
    funsig_t *sig; // signature of the function being called
    bool (*append_rest)(void *, rsymbol_t *, void *);
    bool (*set_arg)(void *, argdesc_t *, void *);
} call_ctx_t;
```

The `set_argbit` helper records the assignment of an argument at position `i` by setting the corresponding bit in `.posbits`. In case it's `omit`ted, the corresponding bit in `.argbits` will be left unset.

⟨*set_argbit*⟩≡
```
static inline bool set_argbit(call_ctx_t *ctx, int i, bool omit)
{
    if(i >= sizeof(argbits_t)*8)
        return false;
    if(!omit)
        ctx->argbits |= 1<<i;
    ctx->posbits |= 1<<i;
    return true;
}
```

A named actual argument is matched by `call_match_kwd`. When a formal argument is found with the same `name`, the corresponding position `i` is marked occupied, and the value `val` assigned by `.set_arg`. Otherwise it's added to the rest vector by `.append_rest`.

⟨*call_match_kwd*⟩≡
```
bool call_match_kwd(void *ptr, rsymbol_t *name, void *val)
{
    call_ctx_t *ctx = ptr;
    int i;
    argdesc_t *arg = get_kwd_arg(ctx->sig, name, &i);

    if(!arg)
        return ctx->append_rest(ctx, name, val);
    return set_argbit(ctx, i, false) && ctx->set_arg(ctx, arg, val);
}
```

Given a `name`, the `get_kwd_arg` helper finds the `argdesc_t` with the same `.name` in the function signature `sig`, passing back its index in `*iptr`. If none exist, it returns `NULL`.

⟨*get_kwd_arg*⟩≡
```
static argdesc_t *get_kwd_arg(funsig_t *sig, rsymbol_t *name, int *iptr)
{
    for(int i = 0; i < sig->nargs; i++)
    {
        argdesc_t *arg = &sig->args[i];

        if(arg->name == name)
        {
            *iptr = i;
            return arg;
        }
    }
    return NULL;
}
```

A positional actual argument is matched by `call_match_pos`. It supplies a value for the first formal argument that hasn't already been assigned, at position `i`. If none remain, it's added to the rest vector.

⟨*call_match_pos*⟩≡
```
bool call_match_pos(void *ptr, void *val)
{
    call_ctx_t *ctx = ptr;
    int i = ffs(~ctx->posbits)-1;
    argdesc_t *arg = get_pos_arg(ctx->sig, i);

    if(!arg)
        return ctx->append_rest(ctx, NULL, val);
    return set_argbit(ctx, i, false) && ctx->set_arg(ctx, arg, val);
}
```

If the signature **sig** contains "...", it must be the last argument, and will be provided specially. It can't be given a value by positional matching, so is ignored by the **get_pos_arg** helper.

⟨*get_pos_arg*⟩≡
```
static argdesc_t *get_pos_arg(funsig_t *sig, int i)
{
    if(i < 0 || i >= nargs_excluding_rest(sig))
        return NULL;
    return &sig->args[i];
}
```

The number of positional arguments in a signature containing "..." is one fewer than **.nargs**.

⟨*nargs_excluding_rest*⟩≡
```
static inline int nargs_excluding_rest(funsig_t *sig)
{
    return sig->has_rest ? sig->nargs - 1 : sig->nargs;
}
```

An omitted actual argument is signalled with **call_match_omit**. It matches the first unassigned formal argument, like a positional, but does not set **.argbits** or supply a value to the **.set_arg** callback. A **NULL** value is added to the rest vector, should the argument be surplus to requirements.

⟨*call_match_omit*⟩≡
```
bool call_match_omit(void *ptr)
{
    call_ctx_t *ctx = ptr;
    int i = ffs(~ctx->posbits)-1;
    bool r = true;

    if(i < 0)
        return false;
    if(i >= nargs_excluding_rest(ctx->sig))
        r = ctx->append_rest(ctx, NULL, NULL);
    return r & set_argbit(ctx, i, true);
}
```

When the rest vector "..." is supplied as an actual argument to a call, the **call_match_rest** matcher will match its elements as arguments. This entails a recursive invocation of **call_sequence**.

⟨*call_match_rest*⟩≡
```
bool call_match_rest(void *ptr, void **args, rsymbol_t **names, int nargs)
{
    return call_sequence(ptr, args, names, nargs, no_match_rest,
                         call_match_kwd, call_match_pos, call_match_omit);
}
```

Rest vectors do not nest. In this case, the **no_match_rest** matcher returns **false** to signal its failure.

⟨*no_match_rest*⟩≡
```
bool no_match_rest(void *ptr, void *rest)
    { return false; }
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
```

⟨*rt/call.h*⟩≡
⟨*call_ctx_t*⟩
```

// first arg is a call_ctx_t *
// the void * decl is so you can use them as callbacks with call_sequence
bool call_match_kwd(void *ptr, rsymbol_t *name, void *val);
bool call_match_pos(void *ptr, void *val);
bool call_match_omit(void *ptr);
bool call_match_rest(void *ptr, void **args, rsymbol_t **names, int nargs);
bool no_match_rest(void *ptr, void *rest);

bool call_sequence(void *ptr, void **args, rsymbol_t **names, int nargs,
                   bool (*rest_fn)(void *, void *),
                   bool (*kwd_fn)(void *, rsymbol_t *, void *),
                   bool (*pos_fn)(void *, void *),
                   bool (*omit_fn)(void *));
```

Chapter 23

Scalars

The runtime provides three *scalar* types – booleans, integers, and double-precision floating point reals. Values of these types may be stored in reference objects called *boxes*, in vectors and arrays. They correspond to data types provided by the machine, and can be operated upon directly by code which has been specialised by the compiler.

⟨*rt/scalar.c*⟩≡
  ⟨*includes*⟩
  ⟨*booleans*⟩
  ⟨*ints*⟩
  ⟨*doubles*⟩
  ⟨*scal_descs*⟩
  ⟨*operations*⟩
  ⟨*boxes*⟩
  ⟨*rscal_install*⟩
  ⟨*rt_install_scalar_types*⟩

23.1 Scalar Types

An instance of a `scaldesc_t` structure describes a scalar type. The `.size` field contains the size of a single machine scalar of that type, in bytes. The `.width` callback returns the number of characters a value will require when printed; this is passed to `.print` to correctly format the output. `.printsz` bytes of storage will be allocated beforehand, in which extra formatting information may be stored.

The type will be installed in the global environment as `.name`.

⟨*scaldesc_t*⟩≡
```
typedef struct scaldesc
{
    char *name;
    size_t size;
    int (*width)(const void *, int, void *);
    void (*print)(FILE *, const void *, int, void *);
    size_t printsz;
} scaldesc_t;
```

Code generation and certain builtin functions have similar (but not identical) behaviour for values of scalar type (Section 26.8). Their differences are parameterised by tables indexed by `scalar_code`.

⟨*scalar_code*⟩≡
```
typedef enum
{
    SC_BOOLEAN = 0,
    SC_INT,
    SC_DOUBLE,
    SC_MAX
} scalar_code;
```

The statically initialised `scal_descs` array contains `scaldesc_t`s for all scalar types; only `double`s require extra storage during printing, of type `dwidth_t`.

⟨*scal_descs*⟩≡
```
const scaldesc_t scal_descs[] = {
    [SC_BOOLEAN] = {
        .name = "boolean",
        .size = sizeof(rboolean_t),
        .print = boolean_print,
        .width = boolean_width,
    },
    [SC_INT] = {
        .name = "int",
        .size = sizeof(rint_t),
        .print = int_print,
        .width = int_width,
    },
    [SC_DOUBLE] = {
        .name = "double",
        .size = sizeof(rdouble_t),
        .print = double_print,
        .width = double_width,
        .printsz = sizeof(dwidth_t)
    }
};
```

A type of `SCALAR` kind will point its `.scal` field to an element of this array. Since these are in `scalar_code` order, `rscal_code` can compute a type's code by pointer subtraction.

⟨*rscal_code*⟩≡
```
static inline scalar_code rscal_code(const rtype_t *typ)
    { return (scalar_code)(typ->scal - scal_descs); }
```

The `.size` field has an accessor for convenience.

⟨*rscal_size*⟩≡
```
static inline size_t rscal_size(const rtype_t *typ)
    { return typ->scal->size; }
```

Scalars can undergo "arithmetic promotion". A scalar value of a type with a lower `scal_code` may be implicitly converted to a type with a higher code, if necessary. `rscal_promote` is defined on types, and returns the greater of its two arguments in code order (assuming they are both `SCALAR`s.)

⟨*rscal_promote*⟩≡
```
static inline rtype_t *rscal_promote(rtype_t *xt, rtype_t *yt)
    { return (rscal_code(xt) > rscal_code(yt)) ? xt : yt; }
```

23.1.1 Boxes

Where a scalar value is to be manipulated by code that expects a reference object, it will be *boxed* – stored into one freshly allocated for the purpose.

⟨*boxes*⟩≡
  ⟨*r_box_create*⟩
  ⟨*r_box*⟩
  ⟨*r_unbox*⟩

This allocation is done by `r_box_create`. The box comprises an `robject_t` header followed by one scalar of the given type.

⟨*r_box_create*⟩≡

```
robject_t *r_box_create(rtype_t *typ)
{
    assert(rtype_is_scalar(typ));
    return gc_alloc(typ, sizeof(robject_t) + rscal_size(typ));
}
```

The `BOXPTR` macro returns a pointer to the contents of a box. The accessor macro `UNBOX` can be used to access (fetch or store) the value stored there, if its type is known.

⟨*box accessors*⟩≡

```
#define UNBOX(typ, box) *((typ *)BOXPTR(box))
#define BOXPTR(box) ((uint8_t *)(box) + sizeof(robject_t))
```

Otherwise, the functions `r_box` and `r_unbox` may be used. The former creates a new box, into which it copies the given value.

⟨*r_box*⟩≡

```
robject_t *r_box(rtype_t *typ, const void *value)
{
    robject_t *box = r_box_create(typ);
    memcpy(BOXPTR(box), value, rscal_size(typ));
    return box;
}
```

The latter copies the value contained in the given box to a destination address in memory (probably a stack slot, vector/array element, or `rvalue_union_t`.)

⟨*r_unbox*⟩≡

```
void r_unbox(void *dest, robject_t *src)
{
    rtype_t *typ = r_typeof(src);
    assert(rtype_is_scalar(typ));
    memcpy(dest, BOXPTR(src), rscal_size(typ));
}
```

23.1.2 Operations

⟨*operations*⟩≡
  ⟨*rscal_print*⟩
  ⟨*rscal_hash*⟩
  ⟨*rscal_equal*⟩
  ⟨*scal_ops*⟩

Scalars of different types have different printed representations, so `rscal_print` dispatches to the type's `scal.print` callback.

This takes the destination `FILE *` and (a pointer to) the value itself, but also the desired width of the output in characters; which allows us to print, for example, the contents of a vector or array with all elements having equal width.

The `scal.width` callback returns either the width of the given value or the previous width specification, whichever is larger. (there are no previous values here, so we pass zero.)

⟨*rscal_print*⟩≡
```
static void rscal_print(FILE *fp, const void *ptr)
{
    rtype_t *typ = r_typeof(ptr);
    void *data = scalprint_init(typ->scal);
    int width = typ->scal->width(BOXPTR(ptr), 0, data);

    typ->scal->print(fp, BOXPTR(ptr), width, data);
}
```

If more than a single `int` is needed to store a type's width specification, `scal.printsz` will be nonzero. `scalprint_init` will allocate this amount of memory from the stack; it can then be passed through `scal.width` to `scal.print`.

⟨*scalprint_init*⟩≡
```
#define scalprint_init(scal) memset(alloca(scal->printsz), 0, scal->printsz)
```

The hash of a scalar value is the hash of its representation in memory and the hash of its type.

⟨*rscal_hash*⟩≡
```
static uint32_t rscal_hash(const void *ptr)
{
    rtype_t *typ = r_typeof(ptr);
    return hash_code_seed(BOXPTR(ptr), rscal_size(typ), r_hash(typ));
}
```

Equality is also defined on the value's representation, and returns a binary result. This is used mainly by the system – arithmetic `==` takes `NA` and `NaN` semantics into account, and can also return `NA` (Appendix B).

⟨*rscal_equal*⟩≡
```
static bool rscal_equal(const void *x, const void *y)
{
    rtype_t *typ = r_typeof(x);
    return !memcmp(BOXPTR(x), BOXPTR(y), rscal_size(typ));
}
```

Scalars (unsurprisingly) don't refer to other objects, so don't require `ops.gc` or `ops.free` callbacks.

⟨*scal_ops*⟩≡
```
static const typeops_t scal_ops = {
    .hash = rscal_hash,
    .equal = rscal_equal,
    .print = rscal_print
};
```

23.2 Booleans

⟨*booleans*⟩≡
  ⟨*boolean_width*⟩
  ⟨*boolean_print*⟩

A `boolean` scalar can take on one of three values – `true`, `false` or `NA`. It's stored in a byte, since the underlying machine has byte-addressable memory (vectors or arrays of booleans admit a packed bit representation, but this optimisation has not been done.)

The `if` expression expects its predicate to evaluate to a non-`NA` boolean.

The `NA` marker for this type has a representation of all-bits-one, but could be any other distinct value.

⟨*rboolean_na*⟩≡
```
static const rboolean_t rboolean_na = ~0;
```

The printed representation of a boolean is either "`true`", "`false`" or "`NA`".

⟨*boolean_width*⟩≡
```
static int boolean_width(const void *ptr, int width, void *data)
{
    rboolean_t v = *(rboolean_t *)ptr;
    int w = (v == rboolean_na) ? 2 : 5-v;
    return max(width, w);
}
```

The output string is right-justified (padded with spaces) in a field of the given `width`.

⟨*boolean_print*⟩≡
```
static void boolean_print(FILE *fp, const void *ptr, int width, void *data)
{
    rboolean_t v = *(rboolean_t *)ptr;
    if(v == rboolean_na)
        fprintf(fp, "%*s", width, "NA");
    else
        fprintf(fp, "%*s", width, v ? "true" : "false");
}
```

23.3 Integers

The `integer` scalar type is the machine integer type, except the minimum representable value is taken as the `NA` marker. It is 32 bits in size (under the ILP-32 and LP-64 data models.)

⟨*ints*⟩≡
  ⟨*int_width*⟩
  ⟨*int_print*⟩

⟨*rint_na*⟩≡
```
static const rint_t rint_na = INT_MIN,
    rint_min = INT_MIN+1,
    rint_max = INT_MAX;
```

Computing the width of a non-`NA` integer is a matter of accounting for the optional leading negative sign, then counting the decimal digits in the absolute value. An integer `NA` also prints as "`NA`".

⟨*int_width*⟩≡
```
static int int_width(const void *ptr, int width, void *data)
{
    rint_t v = *(rint_t *)ptr;
    int w;
    if(v == rint_na)
        w = 2;
    else
        w = (v < 0) + floor(log10(abs(v))) + 1;
    return max(width, w);
}
```

As with booleans, the output is right-justified in its field.

⟨*int_print*⟩≡
```
static void int_print(FILE *fp, const void *ptr, int width, void *data)
{
    rint_t v = *(rint_t *)ptr;
    if(v == rint_na)
        fprintf(fp, "%*s", width, "NA");
    else
        fprintf(fp, "%*d", width, v);
}
```

23.4 Doubles

The `double` scalar type is the machine double-precision floating-point type. In IEC 60559 format, it's always 64 bits in size.

We don't make use of floating-point exceptions; all NaNs are quiet.

⟨*doubles*⟩≡
⟨*is_na*⟩
⟨*dwidth_t*⟩
⟨*double_width*⟩
⟨*double_print*⟩

Since there are a number of representations reserved as NaNs, we take one for our `NA` marker.

⟨*rdouble_na*⟩≡
```
static const rdouble_t rdouble_na = __builtin_nan("0xdeadbeef"),
    rdouble_min = DBL_MIN,
    rdouble_max = DBL_MAX;
```

`is_na` checks for this value with bitwise equality, as comparisons between reals are ordered (and would always return `false`.)

⟨*is_na*⟩≡
```
static inline bool is_na(rdouble_t val)
{
    const uint64_t *pval = (uint64_t *)&val, *pna = (uint64_t *)&rdouble_na;
    return *pval == *pna;
}
```

When computing the printed width of a value of `double` scalar type, a preallocated `dwidth_t` structure is used to track the number of digits before and after the decimal point.

⟨*dwidth_t*⟩≡
```
typedef struct
{
    unsigned short dec, prec;
} dwidth_t;
```

The algorithm is a simplified version of that used by R, implemented by the functions `scientific` and `formatReal` (R Core Team, 2016, src/main/format.c).

⟨*double_width*⟩≡
```
static int double_width(const void *ptr, int width, void *data)
{
    rdouble_t val = *(rdouble_t *)ptr;
    dwidth_t *dwidth = data;
    int dec = 0, prec = 0;
    ⟨compute width⟩
    ⟨update width⟩
}
```

`NA`, `NaN`s and `Inf`inities have fixed-width string representations.

⟨*compute width*⟩≡
```
if(is_na(val))
    return max(width, 2);
else if(!isfinite(val))
    return max(width, 3 + (signbit(val) != 0));
else
{
    ⟨negative⟩
    ⟨non-fractional digits⟩
    ⟨shift and round⟩
    ⟨fractional digits⟩
    ⟨finish⟩
}
```

The optional leading negative sign is accounted for. To ensure correct rounding in the following code, we take the absolute value of the input `val`.

⟨*negative*⟩≡
```
dec = (signbit(val) != 0);
val = fabs(val);
```

The number 10^n (for $n \geq 0$) prints with $n + 1$ digits. If the input is larger than one, taking the logarithm base 10 and rounding down gives us n. After the addition, `dec` contains the number of digits before the decimal point. Subtracting the integral part of `val` leaves the fractional part, upon which we continue to operate.

⟨*non-fractional digits*⟩≡
```
if(val > 1.0)
  dec += floor(log10(val));
dec++;
val -= floor(val);
```

The option `.print_digits` allows the user to specify the maximum number of digits they want printed after the decimal point. Dividing by 10^-n shifts the decimal point right by n digits. Rounding `val` to a whole number then discards anything beyond those digits which the user requested.

⟨*shift and round*⟩≡
```
val /= exp10(-opt.print_digits);
val = nearbyint(val);
```

We don't want to waste output on trailing zeros. Counting digits down from the maximum, we shift the decimal point left until a fractional part is found (or we run out of digits). Undoing the last decrement then leaves `prec` containing the number of digits to print after the decimal point.

⟨*fractional digits*⟩≡
```
for(prec = opt.print_digits;
    prec >= 0 && val == floor(val);
    prec--, val /= 10);
prec++;
```

To distinguish `double`s from `int`s, at least one fractional digit is always printed.

⟨*finish*⟩≡
```
prec = max(prec, 1);
//printf("= dec %d, prec %d\n", dec, prec);
```

The two counts are accumulated separately. The total width of the field is their sum (plus one for the decimal point.)

⟨*update width*⟩≡
```
dwidth->dec = max(dwidth->dec, dec);
dwidth->prec = max(dwidth->prec, prec);
return max(width, dwidth->dec + dwidth->prec + 1);
```

`NA` has the same printed representation as for the other scalar types. All other values are right-justified in `width` columns, with precision `dwidth.prec`.

⟨*double_print*⟩≡
```
static void double_print(FILE *fp, const void *ptr, int width, void *data)
{
    rdouble_t v = *(rdouble_t *)ptr;
    dwidth_t *dwidth = data;

    assert(dwidth);
    if(is_na(v))
        fprintf(fp, "%*s", width, "NA");
    else
        fprintf(fp, "%*.*f", width, dwidth->prec, v);
}
```

23.5 Initialisation

The convenience function `rtype_scal_init` creates and installs a type of `SCALAR` kind from the specified scalar descriptor.

⟨*rscal_install*⟩≡
```
static rtype_t *rtype_scal_init(scalar_code code)
{
    const scaldesc_t *scal = &scal_descs[code];
    rtype_t *typ = rtype_create(RT_SCALAR, (void *)scal,
                                &scal_ops, scal->name);
    rtype_install(typ, scal->name);
    return typ;
}
```

`rt_install_scalar_types` is called during runtime initialisation, and also defines the boolean constants `true` and `false`, and a double constant `Inf`. `NA` is defined as a boolean, since it can be promoted to any type higher in the order as necessary.

⟨*rt_install_scalar_types*⟩≡
```
rtype_t *r_type_int, *r_type_double, *r_type_boolean;
void rt_install_scalar_types()
{
    r_type_double = rtype_scal_init(SC_DOUBLE);
    r_type_int = rtype_scal_init(SC_INT);
    r_type_boolean = rtype_scal_init(SC_BOOLEAN);

    r_defscalar("true", r_type_boolean,
                (rvalue_union_t) { .boolean = true } );
    r_defscalar("false", r_type_boolean,
                (rvalue_union_t) { .boolean = false } );
    r_defscalar("NA", r_type_boolean,
                (rvalue_union_t) { .boolean = rboolean_na } );
    r_defscalar("Inf", r_type_double,
                (rvalue_union_t) { .dfloat = HUGE_VAL } );
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
```

⟨*rt/scalar.h*⟩≡
```
⟨scalar_code⟩
⟨scaldesc_t⟩
⟨box accessors⟩
⟨rint_na⟩
⟨rdouble_na⟩
⟨rboolean_na⟩
⟨externs⟩
⟨prototypes⟩
⟨rscal_code⟩
⟨rscal_size⟩
⟨rscal_promote⟩
⟨scalprint_init⟩
```

⟨*externs*⟩≡
```
extern rtype_t *r_type_int, *r_type_double, *r_type_boolean;
extern const scaldesc_t scal_descs[];
```

⟨*prototypes*⟩≡

```
robject_t *r_box_create(rtype_t *typ);
robject_t *r_box(rtype_t *typ, const void *value);
void r_unbox(void *dest, robject_t *src);
void rt_install_scalar_types();
```

Chapter 24

Vectors and Arrays

The one-dimensional *vector* and multidimensional *array* comprise the collection types provided by the runtime. They are not as fundamental as they would be in a genuine *collection-oriented* language (Sipelstein and Blelloch, 1990).

⟨*rt/vector.c*⟩≡
  ⟨*includes*⟩
  ⟨*buffers*⟩
  ⟨*printing*⟩
  ⟨*vectors*⟩
  ⟨*arrays*⟩
  ⟨*types*⟩
  ⟨*rtype_vec_init*⟩
  ⟨*rt_install_vec_types*⟩

24.1 Containers

Vectors and arrays augment a base *buffer* with additional fields. The `rbuf_t` structure is the first member in these objects. The `.elts` field points to an allocation of unmanaged memory containing `.length` elements.

⟨*rbuf_t*⟩≡

```
typedef struct
{
    robject_t base;
    int length;
    void *elts;
} rbuf_t;
```

The `rvec_elts` and `rvec_len` accessors are macros for convenient use on either kind of container (despite the prefix.)

⟨*rvec accessors*⟩≡

```
#define rvec_elts(_b) ((_b)->buf.elts)
#define rvec_len(_b)  ((_b)->buf.length)
```

24.1.1 Types

An object holding a buffer is of some *container type*. This type is either a `VECTOR` or an `ARRAY`, and holds the buffer's *element type* in its `.elt` field.

⟨*elt_type*⟩≡
```
static inline rtype_t *elt_type(const rbuf_t *buf)
{
    rtype_t *typ = r_typeof(buf);
    assert(rtype_is_container(typ) && typ->elt);
    return typ->elt;
}
```

Only objects with types that are subtypes (Section 20.5) of the element type can be held in the buffer (it's homogeneous just when this is true of the element type alone, as `r_subtype(x, x)` is true for all `x`.)

⟨*types*⟩≡
⟨*cntrdesc_t*⟩
⟨*cntr_type_create*⟩
⟨*cntr_type_from_spec*⟩
⟨*cntr_descs*⟩
⟨*type_creates*⟩

A `cntrdesc_t` structure describes a kind of container types. `.cons` is the constructor descriptor; `.ptype` holds the address of the type which is created from it. The `.ops` callbacks are used for instances of (this kind of) container type.

⟨*cntrdesc_t*⟩≡
```
typedef struct
{
    consdesc_t cons;
    rtype_t **ptype;
    const typeops_t *ops;
} cntrdesc_t;
```

Vectors and arrays share a `cons.from_spec` callback.

⟨*cntr_descs*⟩≡
```
static const cntrdesc_t cntr_vec = {
    .cons = {
        .from_spec = cntr_type_from_spec,
        .kind = RT_VECTOR
    },
    .ptype = &r_type_vector,
    .ops = &vec_ops
};
static const cntrdesc_t cntr_arr = {
    .cons = {
        .from_spec = cntr_type_from_spec,
        .kind = RT_ARRAY
    },
    .ptype = &r_type_array,
    .ops = &arr_ops
};
```

Given a `cntrdesc_t` and an element type, creating an instance of that kind is simple.

⟨*cntr_type_create*⟩≡
```
rtype_t *cntr_type_create(const cntrdesc_t *cntr, rtype_t *etyp)
{
    return rtype_create(cntr->cons.kind, etyp, cntr->ops, NULL);
}
```

Creating an instance from a type specifier is not much more complicated. Since the `consdesc_t` is contained within a `cntrdesc_t`, the address of the latter can be recovered from the former.

If no arguments are provided, the type constructor itself is returned, as it's a supertype of all its instances. Otherwise, the single argument specifies the element type.

⟨*cntr_type_from_spec*⟩≡
```
static rtype_t *cntr_type_from_spec(const consdesc_t *cons,
                                    unsigned nargs, ast_t *args)
{
    const cntrdesc_t *cntr = container_of(cons, cntrdesc_t, cons);

    if(nargs == 0)
        return *cntr->ptype;
    else if(nargs == 1)
    {
        rtype_t *etyp = rtype_from_spec(&args[0]);
        if(etyp)
            return cntr_type_create(cntr, etyp);
    }
    return NULL;
}
```

Wrappers allow convenient creation of vector and array types.

⟨*type_creates*⟩≡
```
rtype_t *rvec_type_create(rtype_t *etyp)
    { return cntr_type_create(&cntr_vec, etyp); }
rtype_t *rarr_type_create(rtype_t *etyp)
    { return cntr_type_create(&cntr_arr, etyp); }
```

24.1.2 Buffers

⟨*buffers*⟩≡
⟨*buf_scal_equal*⟩
⟨*buf_obj_equal*⟩
⟨*rbuf_equal*⟩
⟨*buf_scal_hash*⟩
⟨*buf_obj_hash*⟩
⟨*rbuf_hash*⟩
⟨*rbuf_gc*⟩
⟨*rbuf_free*⟩
⟨*rbuf_create*⟩

Creating a buffer requests a managed object (`sz` bytes large, of container type `typ`) and enough unmanaged memory for `length` elements. Buffers may not be resized in-place.

⟨*rbuf_create*⟩≡
```
static inline void *rbuf_create(rtype_t *typ, unsigned length, size_t sz)
{
    assert(rtype_is_container(typ));
    void *elts;
    size_t vecsz = length * rtype_eltsz(typ->elt);
    rbuf_t *buf = gc_alloc_vec(typ, sz, vecsz, &elts);

    buf->elts = elts;
    buf->length = length;
    return buf;
}
```

Two buffers are equal if they have equal element type, length, and contents.

⟨*rbuf_equal*⟩≡
```
static inline bool rbuf_equal(const rbuf_t *x, const rbuf_t *y)
{
    rtype_t *xt = elt_type(x), *yt = elt_type(y);

    if(xt != yt)
        return false;
    if(x->length != y->length)
        return false;
    return rtype_is_scalar(xt)
        ? buf_scal_equal(x, y, xt)
        : buf_obj_equal(x, y);
}
```

As an optimisation, scalar buffers can have their memory compared directly, since that's how scalar equality is defined (Subsection 23.1.2).

⟨*buf_scal_equal*⟩≡
```
static inline bool
buf_scal_equal(const rbuf_t *x, const rbuf_t *y, rtype_t *typ)
    { return !memcmp(x->elts, y->elts, x->length * rscal_size(typ)); }
```

Buffers containing objects must compare the corresponding elements with `r_equal`.

⟨*buf_obj_equal*⟩≡
```
static inline bool buf_obj_equal(const rbuf_t *x, const rbuf_t *y)
{
    robject_t **vx = x->elts, **vy = y->elts;
    for(int i=0; i < x->length; i++)
        if(!r_equal(vx[i], vy[i]))
            return false;
    return true;
}
```

The hash of a buffer is the hash of its elements and the hash of its type.

⟨*rbuf_hash*⟩≡
```
static inline uint32_t rbuf_hash(const rbuf_t *buf)
{
    uint32_t hash = r_hash(r_typeof(&buf->base));
    rtype_t *etyp = elt_type(buf);

    return rtype_is_scalar(etyp)
        ? buf_scal_hash(buf, hash, etyp)
        : buf_obj_hash(buf, hash);
}
```

As with equality, a buffer of scalars can have its memory hashed directly.

⟨*buf_scal_hash*⟩≡
```
static inline uint32_t
buf_scal_hash(const rbuf_t *buf, uint32_t hash, rtype_t *etyp)
{
    size_t vecsz = buf->length * rtype_eltsz(etyp);
    return hash_code_seed(buf->elts, vecsz, hash);
}
```

Each element in a buffer of objects is hashed, their values contributing to the result.

⟨*buf_obj_hash*⟩≡
```
static inline uint32_t buf_obj_hash(const rbuf_t *buf, uint32_t hash)
{
    robject_t **elts = buf->elts;
    for(int i = 0; i < buf->length; i++)
        hash = hash_roll(hash, r_hash(elts[i]));
    return hash;
}
```

Reference objects contained in a buffer must be marked during garbage collection.

⟨*rbuf_gc*⟩≡
```
static inline void rbuf_gc(rbuf_t *buf)
{
    if(!rtype_is_scalar(elt_type(buf)))
    {
        robject_t **elts = buf->elts;
        for(int i = 0; i < buf->length; i++)
            gc_mark(elts[i]);
    }
}
```

A buffer (of nonzero length) has unmanaged memory allocated to hold its elements, so is responsible for its deallocation upon destruction.

⟨*rbuf_free*⟩≡
```
static inline void rbuf_free(rbuf_t *buf)
{
    if(buf->elts)
        gc_free_vec(buf->elts, buf->length * rtype_eltsz(elt_type(buf)));
}
```

Vectors and arrays may be *named* – that is, contain human-readable symbolic metadata. This isn't stored in the buffer, so depends on the collection's type.

⟨*is_named*⟩≡
```
static inline bool is_named(const robject_t *obj)
{
    rtype_t *type = r_typeof(obj);
    if(rtype_is_vector(type))
        return rvec_is_named((rvector_t *)obj);
    else if(rtype_is_array(type))
        return rarr_is_named((rarray_t *)obj);
    return false;
}
```

24.2 Vectors

A vectors is a dense, ordered, fixed-length collection of elements. The `rvector_t` structure begins with a buffer having `VECTOR` type.

⟨*rvector_t*⟩≡
```
typedef struct rvector_s rvector_t;
typedef struct rvector_s
{
    rbuf_t buf;
    rvector_t *names;
} rvector_t;
```

When the `.names` field is non-`NULL`, it points to a vector of symbols, each element of which names the corresponding element of `.buf.elts`.

⟨*rvec_is_named*⟩≡
```
static inline bool rvec_is_named(const rvector_t *vec)
    { return vec->names != NULL; }
```

⟨*vectors*⟩≡
```
⟨rvec_create⟩
⟨rvec_add_names⟩
⟨rvec_free⟩
⟨rvec_gc⟩
⟨rvec_hash⟩
⟨rvec_equal⟩
⟨rvec_print⟩
⟨vec_ops⟩
```

Vectors are created without element names.

⟨*rvec_create*⟩≡
```
rvector_t *rvec_create(rtype_t *typ, unsigned length)
{
    assert(rtype_is_vector(typ));
    rvector_t *vec = rbuf_create(typ, length, sizeof(rvector_t));
    vec->names = NULL;
    return vec;
}
```

`rvec_add_names` initialises and attaches a name vector to `vec`, returning it to the caller (note that it allocates memory; since this may invoke the garbage collector, `vec` needs to be reachable from some GC root.)

⟨*rvec_add_names*⟩≡
```
rvector_t *rvec_add_names(rvector_t *vec)
{
    int len = rvec_len(vec);
    assert(!vec->names);
    vec->names = rvec_create(r_type_vec_symbol, len);
    memset(rvec_elts(vec->names), 0, len * sizeof(rsymbol_t *));
    return vec->names;
}
```

The buffer's hash is combined with the name vector's, if the latter is present.

⟨*rvec_hash*⟩≡
```
static uint32_t rvec_hash(const void *ptr)
{
    const rvector_t *vec = ptr;
    uint32_t hash = rbuf_hash(&vec->buf);
    if(vec->names)
        return hash_roll(hash, r_hash(vec->names));
    return hash;
}
```

The elements and element names of two vectors must be equal for them to be considered equal.

⟨*rvec_equal*⟩≡
```
static bool rvec_equal(const void *xp, const void *yp)
{
    const rvector_t *x = xp, *y = yp;
    if(!rbuf_equal(&x->buf, &y->buf)
       || rvec_is_named(x) != rvec_is_named(y))
        return false;
    return r_equal(x->names, y->names);
}
```

Vectors have a slightly more verbose printed representation than in R. The first line is a header; the elements follow on the next, though the exact format differs depending on whether or not they are scalars (Subsection 24.4.2).

⟨*rvec_print*⟩≡
```
static void rvec_print(FILE *fp, const void *ptr)
{
    const rvector_t *vec = ptr;
    rtype_t *etyp = elt_type(&vec->buf);
    rsymbol_t **names = vec->names ? rvec_elts(vec->names) : NULL;

    print_vec_hdr(fp, vec);
    if(rtype_is_scalar(etyp))
        print_vec_scal(fp, etyp->scal, rvec_elts(vec), names, rvec_len(vec));
    else
        print_vec_obj(fp, rvec_elts(vec), names, rvec_len(vec));
}
```

Aside from (possibly) marking the contents of the buffer, a vector also needs to mark its name vector when one is present.

⟨*rvec_gc*⟩≡
```
static void rvec_gc(void *ptr)
{
    rvector_t *vec = ptr;
    rbuf_gc(&vec->buf);
    gc_mark(vec->names);
}
```

The unmanaged memory holding the buffer's elements needs explicit deallocation.

⟨*rvec_free*⟩≡
```
static void rvec_free(void *ptr)
{
    rbuf_free(ptr);
}
```

⟨*vec_ops*⟩≡
```
static const typeops_t vec_ops = {
    .free = rvec_free,
    .gc = rvec_gc,
    .hash = rvec_hash,
    .equal = rvec_equal,
    .print = rvec_print
};
```

24.3 Arrays

An array is a dense, multi-dimensional collection with a fixed shape. `.buf.type` is of kind `ARRAY`. Elements are stored at `.buf.elts` in column-major order. The extents of the array are recorded in `.shape`, and its dimension in `.rank`.

⟨*rarray_t*⟩≡
```
typedef struct
{
    rbuf_t buf;
    int rank;
    int *shape;
    rvector_t *dimnames;
} rarray_t;
```

If `.dimnames` is non-`NULL`, the array is named. Each (non-`NULL`) element of this vector is itself a vector of names for the indices of the corresponding dimension.

⟨*rarr_is_named*⟩≡
```
static inline bool rarr_is_named(const rarray_t *arr)
    { return arr->dimnames != NULL; }
```

Two arrays *conform* if they have the same dimensions – that is, their ranks and shapes are equal.

⟨*rarr_conform*⟩≡
```
static inline bool rarr_conform(const rarray_t *x, const rarray_t *y)
    { return rarr_shape_conform(x->rank, y->rank, x->shape, y->shape); }
```

⟨*rarr_shape_conform*⟩≡
```
static inline bool rarr_shape_conform(int xr, int yr, int *xs, int *ys)
    { return (xr == yr) && !memcmp(xs, ys, sizeof(int) * xr); }
```

⟨*arrays*⟩≡
⟨*arr_length*⟩
⟨*rarr_create_buf*⟩
⟨*rarr_set_shape*⟩
⟨*rarr_create*⟩
⟨*rarr_add_names*⟩
⟨*rarr_free*⟩
⟨*rarr_gc*⟩
⟨*rarr_hash*⟩
⟨*rarr_equal*⟩
⟨*rarr_print*⟩
⟨*arr_ops*⟩

Creating an array is a matter of creating the underlying buffer and initialising the `.shape` field (these functions are split out so that they are available to the vectorised arithmetic subsystem in Appendix B.)

⟨*rarr_create*⟩≡
```
rarray_t *rarr_create(rtype_t *typ, int rank, int *dims)
{
    assert(rank >= 0);
    int len = arr_length(rank, dims);
    rarray_t *arr = rarr_create_buf(typ, len);
    rarr_set_shape(arr, rank, dims);
    return arr;
}
```

`.buf` must have length equal to the product of the provided dimensions.

⟨*arr_length*⟩≡
```
static int arr_length(int rank, int *dims)
{
    int len = rank > 0 ? 1 : 0;
    for(int i = 0; i < rank; i++)
        len *= dims[i];
    return len;
}
```

As per vectors, arrays are created without dimnames.

⟨*rarr_create_buf*⟩≡
```
rarray_t *rarr_create_buf(rtype_t *typ, int len)
{
    rarray_t *arr = rbuf_create(typ, len, sizeof(rarray_t));

    arr->rank = 0;
    arr->shape = NULL;
    arr->dimnames = NULL;
    return arr;
}
```

The `.shape` of an array is stored out-of-line in unmanaged memory (the provided dimensions are not sanity checked against `.buf.length`; this function is not user-visible.)

⟨*rarr_set_shape*⟩≡
```
void rarr_set_shape(rarray_t *arr, int rank, int *dims)
{
    size_t sz = rank * sizeof(int);
    int *shape = rank > 0 ? xmalloc(sz) : NULL;

    memcpy(shape, dims, sz);
    arr->rank = rank;
    if(arr->shape)
        xfree(arr->shape);
    arr->shape = shape;
}
```

Adding dimnames to an array creates and initialises a new vector, assigning it to `.dimnames` (the note from `rvec_add_names` applies in this case, too.) Each dimension can then have its indices named by setting its corresponding element to a vector of symbols with correct length.

⟨*rarr_add_names*⟩≡
```
rvector_t *rarr_add_names(rarray_t *arr)
{
    assert(!arr->dimnames);
    arr->dimnames = rvec_create(r_type_vec_names, arr->rank);
    memset(rvec_elts(arr->dimnames), 0, arr->rank * sizeof(rvector_t *));
    return arr->dimnames;
}
```

The hash of an array additionally includes the hash of its shape.

⟨*rarr_hash*⟩≡
```
static uint32_t rarr_hash(const void *ptr)
{
    const rarray_t *arr = ptr;
    uint32_t hash = rbuf_hash(&arr->buf);
    hash = hash_code_seed(arr->shape, arr->rank * sizeof(int), hash);
    if(arr->dimnames)
        return hash_roll(hash, r_hash(arr->dimnames));
    return hash;
}
```

Two arrays are equal if their shapes conform, and their elements and dimnames are equal.

⟨*rarr_equal*⟩≡
```
static bool rarr_equal(const void *xp, const void *yp)
{
    const rarray_t *x = xp, *y = yp;

    if(!rarr_conform(x, y)
       || !rbuf_equal(&x->buf, &y->buf)
       || rarr_is_named(x) != rarr_is_named(y))
        return false;
    return r_equal(x->dimnames, y->dimnames);
}
```

Arrays are printed in a similar manner to vectors; the header line is followed by the elements, with a different format for scalars.

⟨*rarr_print*⟩≡
```
static void rarr_print(FILE *fp, const void *ptr)
{
    const rarray_t *arr = ptr;
    rtype_t *etyp = elt_type(&arr->buf);
    rvector_t **dimnames = arr->dimnames ? rvec_elts(arr->dimnames) : NULL;

    print_arr_hdr(fp, arr);
    if(rtype_is_scalar(etyp))
        print_arr_scal(fp, etyp->scal, rvec_elts(arr), dimnames,
                       arr->rank, arr->shape);
    else
        print_arr_obj(fp, rvec_elts(arr), dimnames,
                      arr->rank, arr->shape);
}
```

The dimnames vector must be marked, as well as the elements in the buffer (depending on type.)

⟨*rarr_gc*⟩≡
```
static void rarr_gc(void *ptr)
{
    rarray_t *arr = ptr;
    rbuf_gc(&arr->buf);
    gc_mark(arr->dimnames);
}
```

In addition to the buffer, `.shape` must be deallocated.

⟨*rarr_free*⟩≡
```
static void rarr_free(void *ptr)
{
    rarray_t *arr = ptr;
    rbuf_free(&arr->buf);
    if(arr->shape)
        xfree(arr->shape);
}
```

⟨*arr_ops*⟩≡
```
static const typeops_t arr_ops = {
    .free = rarr_free,
    .gc = rarr_gc,
    .hash = rarr_hash,
    .equal = rarr_equal,
    .print = rarr_print
};
```

24.4 Printing

The printed representations of vectors and arrays take into account element type, presence of (dim)names, and user-specified options.

Various fields (names sometimes, scalars always) are tabulated by computing the maximum printed width of the values in each column, then padding each element with whitespace to ensure horizontal alignment.

⟨*printing*⟩≡
⟨*clip_to_max*⟩
⟨*clip_npieces*⟩
⟨*name_width*⟩
⟨*names_width*⟩
⟨*scalfmt_t*⟩
⟨*scalfmt_init*⟩
⟨*scalfmt_width*⟩
⟨*idx_width*⟩
⟨*print_name*⟩
⟨*printing vectors*⟩
⟨*printing arrays*⟩

Users may specify with `opt.print_max` a maximum number of elements to print. `clip_to_max` returns `true` if its input was reduced due to exceeding this value.

⟨*clip_to_max*⟩≡
```
static inline bool clip_to_max(int *plen)
{
    if(*plen > opt.print_max)
    {
        *plen = opt.print_max;
        return true;
    }
    return false;
}
```

If a 'piece' consists of `piecelen` elements, and there are `*npieces` to print, `clip_npieces` adjusts the latter when the total number of elements exceeds the specified maximum. In this case, it returns `true`.

⟨*clip_npieces*⟩≡
```
static bool clip_npieces(int *npieces, int piecelen)
{
    int len = *npieces * piecelen;
    bool truncated = clip_to_max(&len);

    if(piecelen > 0)
    {
        // divide into pieces
        div_t r = div(len, piecelen);
        *npieces = r.quot + (r.rem > 0);
    }
    return truncated;
}
```

Users may also specify a maximum printed name length with `opt.print_name_max`. `name_width` returns the number of characters in a name, respecting this limit.

⟨*name_width*⟩≡
```
static inline int name_width(rsymbol_t *name)
{
    int l = strlen(r_symstr(name));
    return min(l, opt.print_name_max);
}
```

`names_width` returns the number of characters taken by the longest name of those it's given.

⟨*names_width*⟩≡
```
static int names_width(rsymbol_t **names, int len)
{
    int width = 0;
    for(int i = 0; i < len; i++)
    {
        int w = name_width(names[i]);
        width = max(width, w);
    }
    return width;
}
```

This can be taken into account when computing the width of a column containing names (possibly amongst other things,) so that its elements can be printed in right-justified fields of equal width (which is used as both precision and width specifier, since `name` might need to be truncated.)

⟨*print_name*⟩≡
```
static void print_name(FILE *fp, rsymbol_t *name, int width)
{
    fprintf(fp, "%*.*s", width, width, r_symstr(name));
}
```

An index value (if not otherwise named) is printed as an integer, so its width is that of an equivalent `r_type_int`.

⟨*idx_width*⟩≡
```
static inline int idx_width(int i)
{
    const scaldesc_t *intscal = r_type_int->scal;
    return intscal->width(&i, 0, NULL);
}
```

The maximum printed `.width` can be found for a column (or vector) of scalars, but some types need additional `.data`.

⟨*scalfmt_t*⟩≡
```
typedef struct
{
    int width;
    void *data;
} scalfmt_t;
```

`scalfmt_width` updates `fmt` with just this value, computed over the `len` scalar elements at `eptr`.

⟨*scalfmt_width*⟩≡
```
static inline void scalfmt_width(scalfmt_t *fmt, const scaldesc_t *scal,
                                 uint8_t *eptr, int len)
{
    for(int i = 0; i < len; i++, eptr += scal->size)
        fmt->width = scal->width(eptr, fmt->width, fmt->data);
}
```

Since other values can be printed in the same column, `scalfmt_init` takes an initial value of width as input.

⟨*scalfmt_init*⟩≡
```
#define scalfmt_init(w, s) \
    (scalfmt_t) { .width = (w), .data = scalprint_init(s) }
```

24.4.1 Printing Vectors

⟨*printing vectors*⟩≡
⟨*print_vec_hdr*⟩
⟨*print_vec_scal_elts*⟩
⟨*print_vec_scal_names*⟩
⟨*print_vec_obj*⟩
⟨*print_vec_scal*⟩

The header line of a vector prints as "`vector(type) [length]`".

⟨*print_vec_hdr*⟩≡
```
static void print_vec_hdr(FILE *fp, const rvector_t *vec)
{
    r_print(fp, r_typeof(vec));
    fprintf(fp, " [%d]", rvec_len(vec));
}
```

When the vector doesn't have scalar element type, each element is `r_print`ed on a line of its own. The line is prefixed with a "`name:`" (if the vector has them) or an "`[index]:` " (if not.)

The maximum printed `width` of these prefixes defines the width of the field in which each is right justified. A trailing "`...`" signifies the output was `truncated` at user request.

⟨*print_vec_obj*⟩≡
```
static void print_vec_obj(FILE *fp, robject_t **objs, rsymbol_t **names,
                          int len)
{
    int width = names
        ? names_width(names, len)
        : idx_width(len);
    bool truncated = clip_to_max(&len);

    for(int i = 0; i < len; i++)
    {
        fprintf(fp, "\n");
        if(names)
            print_name(fp, names[i], width);
        else
            fprintf(fp, "[%*d]", width, i + 1);
        fprintf(fp, ": ");
        r_print(fp, objs[i]);
    }

    if(truncated)
        fprintf(fp, "\n ...");
}
```

A vector of scalars prints more compactly, at the cost of some extra work up-front. It is also `truncated` with "`...`" if necessary.

⟨*print_vec_scal*⟩≡
```
static void print_vec_scal(FILE *fp, const scaldesc_t *scal, void *elts,
                           rsymbol_t **names, int len)
{
    uint8_t *eptr = elts;
    bool truncated = clip_to_max(&len);
    ⟨print vector⟩
    if(truncated)
        fprintf(fp, " ...");
}
```

All elements are printed in fields of identical width – equal to the maximum required by any element in the vector. The maximum width of the element names is also taken into account, if they are present.

⟨*print vector*⟩≡
```
scalfmt_t fmt = scalfmt_init(names ? names_width(names, len) : 0, scal);
scalfmt_width(&fmt, scal, elts, len);
```

The printed representation of the vector may span multiple lines. If element names are absent, each begins with "`[index]`", `nch` characters wide, with the index right-justified in a field of `iwidth`.

⟨*print vector*⟩+≡
```
int iwidth = idx_width(len);
int nch = names ? 0 : iwidth + 2;
```

The user option `opt.print_line` specifies the number of characters after which a line should be broken. Until this limit is exceeded, `linech` accumulates the widths of elements (including leading spaces) so as to compute `llen`, the number of elements to print on a single line. `lsz` is the size in bytes of the line's worth of elements (as scalar types differ in size.)

⟨*print vector*⟩+≡
```
int linech = nch, llen = 0;
size_t lsz = scal->size;
for(int i = 0; i < len; i++)
{
    linech += fmt.width + 1;
    if(linech > opt.print_line && i > 0)
        break;
    llen++;
}
lsz *= llen;
```

Each line is printed while advancing the index `i` and pointer `eptr`. `plen` accounts for a trailing partial line if `llen` does not divide `len`.

⟨*print vector*⟩+≡
```
for(int i = 0; i < len; i += llen, eptr += lsz)
{
    int plen = min(llen, len - i);

    fprintf(fp, "\n");
    if(!names)
        fprintf(fp, "[%*d]", iwidth, i + 1);
    print_vec_scal_elts(fp, scal, eptr, plen, fmt);
    if(names)
        print_vec_scal_names(fp, names + i, plen, fmt.width);
}
```

The `scal.print` callback is invoked for each element in the line…

⟨*print_vec_scal_elts*⟩≡
```
static inline void
print_vec_scal_elts(FILE *fp, const scaldesc_t *scal, uint8_t *eptr,
                    int len, scalfmt_t fmt)
{
    for(int i = 0; i < len; i++, eptr += scal->size)
    {
        fprintf(fp, " ");
        scal->print(fp, eptr, fmt.width, fmt.data);
    }
}
```

…and if names are present, each is printed below its corresponding element.

⟨*print_vec_scal_names*⟩≡
```
static inline void print_vec_scal_names(FILE *fp, rsymbol_t **names,
                                        int len, int width)
{
    fprintf(fp, "\n");
    for(int i = 0; i < len; i++)
    {
        fprintf(fp, " ");
        print_name(fp, names[i], width);
    }
}
```

24.4.2 Printing Arrays

⟨*printing arrays*⟩≡
  ⟨*idxs_advance*⟩
  ⟨*idxs_init*⟩
  ⟨*idx_name*⟩
  ⟨*print_idxs*⟩
  ⟨*print_arr_hdr*⟩
  ⟨*print_arr_obj*⟩
  ⟨*print_arr_scal_elts*⟩
  ⟨*slice_width*⟩
  ⟨*print_col_idxs*⟩
  ⟨*print_slice*⟩
  ⟨*print_slices*⟩
  ⟨*print_arr_scal*⟩

An array's header line is similar to a vector's, with shape instead of length: "`array(type) [rows,cols,...]`".

⟨*print_arr_hdr*⟩≡
```
static void print_arr_hdr(FILE *fp, const rarray_t *arr)
{
    r_print(fp, r_typeof(arr));
    fprintf(fp, " [");
    for(int i = 0; i < arr->rank; i++)
    {
        if(i > 0)
            fprintf(fp, ", ");
        fprintf(fp, "%d", arr->shape[i]);
    }
    fprintf(fp, "]");
}
```

To step through an array an element at a time, a set of indices are tracked, one per dimension. `idxs_init` zeros its input, accumulating and returning the number of elements in the array.

⟨*idxs_init*⟩≡
```
static inline int idxs_init(int rank, int *shape, int *idxs)
{
    int len = 1;
    for(int i = 0; i < rank; i++)
    {
        idxs[i] = 0;
        len *= shape[i];
    }
    return len;
}
```

Arrays are column-major, so the leftmost index increases fastest when advancing `idxs` through the array.

⟨*idxs_advance*⟩≡
```
static inline void idxs_advance(int rank, int *shape, int *idxs)
{
    for(int i = 0; i < rank; i++)
    {
        if(++idxs[i] >= shape[i])
            idxs[i] = 0;
        else
            break;
    }
}
```

Like vectors, an array without scalar element type prints each element on its own line, with a prefix indicating its position in the container. The output may be `truncated`.

If `dimnames` is non-`NULL`, each element corresponds to a dimension of the array, and is either `NULL` or an `rvector_t` of index names. In this last case, the names are printed in lieu of numeric indices for that dimension.

⟨*print_arr_obj*⟩≡
```
static void print_arr_obj(FILE *fp, robject_t **objs, rvector_t **dimnames,
                          int rank, int *shape)
{
    int *idxs = alloca(rank * sizeof(int));
    int len = idxs_init(rank, shape, idxs);
    bool truncated = clip_to_max(&len);

    for(int i = 0; i < len; i++)
    {
        fprintf(fp, "\n[");
        print_idxs(fp, idxs, rank, dimnames);
        fprintf(fp, "]: ");
        r_print(fp, objs[i]);
        idxs_advance(rank, shape, idxs);
    }
    if(truncated)
        fprintf(fp, "\n ...");
}
```

The prefix is printed as "`[row,col,...]: `", with each index either a name or number.

⟨*print_idxs*⟩≡
```
static inline void print_idxs(FILE *fp, int *idxs, int rank,
                              rvector_t **dimnames)
{
    for(int i = 0; i < rank; i++)
    {
        int idx = idxs[i];

        if(i > 0)
            fprintf(fp, ",");
        if(dimnames && dimnames[i])
            fprintf(fp, "%s", r_symstr(idx_name(dimnames[i], idx)));
        else
            fprintf(fp, "%d", idx + 1);
    }
}
```

Given an element of `dimnames` and an index in the corresponding dimension, `idx_name` extracts the corresponding name.

⟨*idx_name*⟩≡
```
static inline rsymbol_t *idx_name(rvector_t *names, int i)
    { return ((rsymbol_t **)rvec_elts(names))[i]; }
```

The printed representation of an array of scalars depends on its rank. Arrays of zero rank produce no further output; arrays of rank one can reuse `print_vec_scal`. Arrays of higher rank are printed as a sequence of two-dimensional *slices*.

⟨*print_arr_scal*⟩≡
```
static void print_arr_scal(FILE *fp, const scaldesc_t *scal, void *elts,
                           rvector_t **dimnames, int rank, int *shape)
{
    switch(rank)
    {
    case 0:
        break;
    case 1:
        print_vec_scal(fp, scal, elts,
                       dimnames ? rvec_elts(dimnames[0]) : NULL,
                       shape[0]);
        break;
    case 2:
        print_slice(fp, scal, elts, dimnames, shape);
        break;
    default:
        print_slices(fp, scal, elts, dimnames, rank, shape);
        break;
    }
}
```

If more than one slice is to be printed, they are considered as elements of an array of two fewer dimensions, with rank `rrank`, shape `rshape` and length `len`. This reduced-rank array is then stepped through, and each slice (`slicelen` elements in length, `ssz` bytes in size) printed.

If the number of elements exceeds the user's specified maximum, the output will be `truncated`.

A slice has a header line of the form "`[,,idx,idx,...]`", indicating the index of the first element (the leading two indices are omitted as they'll always be `1`.)

⟨*print_slices*⟩≡
```
static void print_slices(FILE *fp, const scaldesc_t *scal, uint8_t *eptr,
                         rvector_t **dimnames, int rank, int *shape)
{
    int rrank = rank - 2, *rshape = shape + 2;
    int slicelen = shape[0] * shape[1];
    int *idxs = alloca(rrank * sizeof(int));
    int len = idxs_init(rrank, rshape, idxs);
    bool truncated = clip_npieces(&len, slicelen);
    size_t ssz = slicelen * scal->size;

    for(int i = 0; i < len; i++, eptr += ssz)
    {
        fprintf(fp, "\n\n[,,");
        print_idxs(fp, idxs, rrank, dimnames ? dimnames + 2 : NULL);
        fprintf(fp, "]");
        print_slice(fp, scal, eptr, dimnames, shape);
```

```
        idxs_advance(rrank, rshape, idxs);
    }
    if(truncated)
        fprintf(fp, "\n ...");
}
```

Convenient synonyms are defined for row and columns counts and names. The slice may be **truncated**; if so, fewer rows will be printed.

⟨*print_slice*⟩≡

```
static void print_slice(FILE *fp, const scaldesc_t *scal, uint8_t *eptr,
                        rvector_t **dimnames, int *shape)
{
    rvector_t *rownames = dimnames ? dimnames[0] : NULL,
              *colnames = dimnames ? dimnames[1] : NULL;
    int nrows = shape[0], ncols = shape[1];
    size_t csz = scal->size * nrows;
    bool truncated = clip_npieces(&nrows, ncols);
    ⟨print slice⟩
    if(truncated)
        fprintf(fp, "\n ...");
}
```

The slice is printed in one or more *chunks* of columns. Each chunk prints as a header line followed by **nrows** rows.

Row indices share a single field width, **riwidth**. This takes **nch** characters – if the rows aren't named, the index prints as "**[idx,]**", so adds 3.

⟨*print slice*⟩≡

```
int riwidth = rownames
    ? names_width(rvec_elts(rownames), nrows)
    : idx_width(nrows);
int nch = rownames ? riwidth : riwidth + 3;
```

Each column has a separate **scalfmt_t**, unlike vectors where one is shared by all elements. A column index contributes its total width to the **.width** of its column, but might have a different field width, stored in **ciwidths**.

⟨*print slice*⟩+≡

```
int *ciwidths = alloca(ncols * sizeof(int));
scalfmt_t *fmts = alloca(ncols * sizeof(scalfmt_t));

for(int i=0; i<ncols; i++)
    fmts[i] = scalfmt_init(0, scal);
slice_width(scal, eptr, fmts, ciwidths, colnames, nrows, ncols);
```

Stepping through the array by column, the index width provides the initial value of `.width` for the scalar elements. If column names are not present, the printed representation "`[,idx]`" adds 3 characters, the same as row indices.

⟨*slice_width*⟩≡
```
static void
slice_width(const scaldesc_t *scal, uint8_t *eptr, scalfmt_t *fmts,
            int *ciwidths, rvector_t *colnames, int nrows, int ncols)
{
    size_t csz = scal->size * nrows;

    for(int i = 0; i < ncols; i++, eptr += csz)
    {
        int iwidth = colnames
            ? name_width(idx_name(colnames, i))
            : idx_width(i + 1);

        ciwidths[i] = iwidth;
        fmts[i].width = colnames ? iwidth : 3 + iwidth;
        scalfmt_width(&fmts[i], scal, eptr, nrows);
    }
}
```

Printing one chunk advances column index `i` to `end`, and initial element pointer `eptr` by the corresponding number of bytes.

⟨*print slice*⟩+≡
```
for(int i = 0, end = 0; i < ncols; eptr += (end - i) * csz, i = end)
{
    ⟨print line⟩
}
```

`linech` records the number of characters printed on a line. Column widths (and separating spaces) are accumulated, updating `end`, until either all have been accounted for or the specified line length is exceeded.

⟨*print line*⟩≡
```
int linech = nch;
for(int j = i; j < ncols; j++)
{
    linech += fmts[j].width + 1;
    if(linech > opt.print_line && j > i)
        break;
    end++;
}
```

Indenting past the row index aligns the chunk header line.

⟨*print line*⟩+≡
```
fprintf(fp, "\n%*s", nch, "");
print_col_idxs(fp, i, end, fmts, ciwidths, colnames);
```

This contains the indices of columns from `i` to `end`. In the absence of names, the bracketed index is right-justified in its field.

⟨*print_col_idxs*⟩≡
```
static inline void print_col_idxs(FILE *fp, int i, int end, scalfmt_t *fmts,
                                  int *ciwidths, rvector_t *colnames)
{
    for(; i < end; i++)
    {
        fprintf(fp, " ");
        if(colnames)
            print_name(fp, idx_name(colnames, i), fmts[i].width);
        else
            fprintf(fp, "%*s[,%*d]",
                    max(fmts[i].width - ciwidths[i] - 3, 0), "",
                    ciwidths[i], i + 1);
    }
}
```

Each row is printed on one line, starting with the row index. `rptr` is set to point to the start of the row, and is stepped by one element, since arrays are stored in column-major order.

⟨*print line*⟩+≡
```
uint8_t *rptr = eptr;
for(int j = 0; j < nrows; j++, rptr += scal->size)
{
    fprintf(fp, "\n");
    if(rownames)
        print_name(fp, idx_name(rownames, j), riwidth);
    else
        fprintf(fp, "[%*d,]", riwidth, j + 1);
    print_arr_scal_elts(fp, scal, rptr, i, end, csz, fmts);
}
```

Elements on a row are printed similarly to those in a vector, except each has its own `scalfmt_t`.

⟨*print_arr_scal_elts*⟩≡
```
static inline void
print_arr_scal_elts(FILE *fp, const scaldesc_t *scal, uint8_t *eptr,
                    int i, int end, int csz, scalfmt_t *fmts)
{
    for(; i < end; i++, eptr += csz)
    {
        fprintf(fp, " ");
        scal->print(fp, eptr, fmts[i].width, fmts[i].data);
    }
}
```

24.5 Initialisation

A convenience function allows the creation and installation of a vector type with a specified element type.

⟨*rtype_vec_init*⟩≡
```
static inline rtype_t *rtype_vec_init(rtype_t *elt_type, const char *name)
{
    rtype_t *type = rvec_type_create(elt_type);
    rtype_install(type, name);
    return type;
}
```

The type constructors `r_type_vector` and `r_type_array` are installed as "`vector`" and "`array`"; instances required by some builtins are saved in `r_type_vec_` globals, and installed under (hopefully) familiar names.

⟨*rt_install_vec_types*⟩≡
```
rtype_t *r_type_vector, *r_type_array,
        *r_type_vec_symbol, *r_type_vec_names, *r_type_vec_object,
        *r_type_vec_boolean, *r_type_vec_int, *r_type_vec_double;

void rt_install_vec_types()
{
    r_type_vector = rtype_cons_init(&cntr_vec.cons, "vector");
    r_type_array = rtype_cons_init(&cntr_arr.cons, "array");
    r_type_vec_object = rtype_vec_init(r_type_object, "list");
    r_type_vec_symbol = rtype_vec_init(r_type_symbol, "names");
    r_type_vec_names = rtype_vec_init(r_type_vec_symbol, "dimnames");
    r_type_vec_boolean = rtype_vec_init(r_type_boolean, "logical");
    r_type_vec_int = rtype_vec_init(r_type_int, "integer");
    r_type_vec_double = rtype_vec_init(r_type_double, "numeric");
}
```

Miscellanea

⟨*includes*⟩≡
```
#include "global.h"
#include "rt/builtin.h"
#include "ast.h"
```

⟨*rt/vector.h*⟩≡
⟨*rbuf_t*⟩
⟨*rvector_t*⟩
⟨*rarray_t*⟩
⟨*externs*⟩
⟨*prototypes*⟩
⟨*elt_type*⟩
⟨*rvec accessors*⟩
⟨*rvec_is_named*⟩
⟨*rarr_is_named*⟩
⟨*is_named*⟩
⟨*rarr_shape_conform*⟩
⟨*rarr_conform*⟩

⟨*externs*⟩≡
```
extern rtype_t *r_type_vector, *r_type_array,
    *r_type_vec_symbol, *r_type_vec_names, *r_type_vec_object,
    *r_type_vec_boolean, *r_type_vec_int, *r_type_vec_double;
```

⟨*prototypes*⟩≡
```
rtype_t *rvec_type_create(rtype_t *elt_type);
rvector_t *rvec_create(rtype_t *typ, unsigned length);
rvector_t *rvec_add_names(rvector_t *vec);

rtype_t *rarr_type_create(rtype_t *etyp);
rarray_t *rarr_create_buf(rtype_t *typ, int length);
void rarr_set_shape(rarray_t *arr, int rank, int *dims);
rarray_t *rarr_create(rtype_t *typ, int rank, int *dims);
rvector_t *rarr_add_names(rarray_t *arr);

void rt_install_vec_types();
```

Chapter 25

Memory Management

Automatic memory management is nearly ubiquitous among high-level languages. The execution overhead is more than outweighed by the advantages it affords; in particular, simplification of user code (Jones et al., 2012).

⟨*rt/gc.c*⟩≡
  ⟨*includes*⟩
  ⟨*bit constants*⟩
  ⟨*size constants*⟩
  ⟨*gc_stats*⟩
  ⟨*interface globals*⟩
  ⟨*chunk globals*⟩
  ⟨*heap*⟩
  ⟨*allocation*⟩
  ⟨*collection*⟩
  ⟨*interface*⟩

Memory used by the runtime can be split into two classes. *Unmanaged* buffers of arbitrary size are handled by the platform `libc`. Some will be owned by other objects, and some will be explicitly handled by the runtime. *Managed* memory contains `robject_ts` Section 19.1, which are the only referents directly visible to user code.

The memory management subsystem, responsible for handling them, is composed of a segregated-fit allocator and a type-accurate mark-sweep garbage collector. This is of deliberately simple design – generational or concurrent collection would improve latency at the cost of implementation complexity.

25.1 Managed Heap

The segregated-fit allocator has one *pool* for each distinct size of object (this set is fixed at compile-time). A pool holds a list of *pages* divided into objects of identical size. A page is itself contained in a larger *chunk* of memory requested from the operating system. Objects on a page which are not currently in use are kept on the page's free list. Empty pages in a chunk are kept on the chunk's free list.

⟨*heap*⟩≡
  ⟨*free_obj_t*⟩
  ⟨*page_t*⟩
  ⟨*pool_t*⟩
  ⟨*chunk_t*⟩
  ⟨*chunks*⟩
  ⟨*pages*⟩
  ⟨*pools*⟩

25.1.1 Chunks

A chunk begins with a `chunk_t` header. After this (and possibly some unusable leftover space), the chunk is divided into pages of constant size.

Each page is described by an entry in the `.pages` array. The `.free_head` field is a list of empty pages in the chunk; `.nlive` counts the number of live objects on all contained pages.

⟨*chunk_t*⟩≡
```
typedef struct
{
    list_t chunk_list;
    list_t free_head;
    unsigned nlive;
    page_t pages[NPAGES];
} chunk_t;
⟨assert size⟩
```

`NPAGES`, the number of pages in a chunk, depends on the size of each page, the size of the chunk, and the size of the header. But the size of the header depends on the number of pages in a chunk. To break this circularity, we fix the amount of memory reserved for the header.

The header is comprised mainly of pointers, so will differ in size between 32- and 64-bit x86 machines (ILP-32 vs LP-64). The data model is visible to the preprocessor via `unistd.h`.

⟨*bit constants*⟩≡
```
#if (defined(_XBS5_ILP32_OFF32) && _XBS5_ILP32_OFF32 != -1)
#define WORDSHIFT 2
#define NRESERVED 3
#elif (defined(_XBS5_LP64_OFF64) && _XBS5_LP64_OFF64 != -1)
#define WORDSHIFT 3
#define NRESERVED 6
#else
#error "platform bitness undefined."
#endif
```

Ideally, chunks would be large enough to amortise syscall overhead but small enough to minimise requested-but-unused memory. In the absence of extensive practical experience, `CHUNKSIZE` is chosen to be 2MiB. `PAGESIZE` is 4KiB (which probably matches the underlying hardware.)

Each chunk then contains `NPAGES` pages available for allocation; with these values, this is 253 (250 on x86-64.)

⟨*size constants*⟩≡
```
#define CHUNKBITS 21
#define CHUNKSIZE (1<<CHUNKBITS)
#define CHUNKMASK (CHUNKSIZE - 1)
#define PAGEBITS  12
#define PAGESIZE  (1<<PAGEBITS)
#define PAGEMASK  (PAGESIZE - 1)
#define NPAGES    ((1<<(CHUNKBITS - PAGEBITS)) - NRESERVED)
```

The chunk header must fit in the area reserved for it. Since its size is available to the compiler, this can be assured.

⟨*assert size*⟩≡
```
static_assert(NRESERVED * PAGESIZE >= sizeof(chunk_t),
              "chunk header too large, reserve more pages.");
```

Chunks are counted, and kept on a global list.

⟨*chunk globals*⟩≡
```
static list_t chunk_head;
static unsigned gc_nchunks = 0;
```

⟨*chunks*⟩≡
```
⟨mmap_chunk⟩
⟨munmap_chunk⟩
⟨mmap_aligned_chunk⟩
⟨chunk_create⟩
⟨chunk_free⟩
⟨chunk_get_page⟩
⟨chunk_for_addr⟩
```

Every chunk is aligned so that, given an internal pointer, it is possible to compute the address of the `chunk_t` header by zeroing the low-order `CHUNKBITS`.

⟨*chunk_for_addr*⟩≡
```
static inline chunk_t *chunk_for_addr(void *obj)
    { return (chunk_t *)((uintptr_t)obj & ~CHUNKMASK); }
```

The allocator will create a chunk when it needs to grow the managed heap. A call to `mmap` requests that the operating system map a new region of memory into the process' address space. A simple wrapper function takes care of unvarying arguments, as well as error handling (such as it is.)

⟨*mmap_chunk*⟩≡
```
static inline void *mmap_chunk(void *hint, size_t sz)
{
    void *r = mmap(hint, sz, PROT_READ|PROT_WRITE,
                   MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);

    if(r == MAP_FAILED)
        fatal("can't allocate %.1fk for chunk: %s.",
              sz / 1024.0, strerror(errno));
    return r;
}
```

(Likewise for `munmap`.)

⟨*munmap_chunk*⟩≡
```
static inline void munmap_chunk(void *ptr, size_t sz)
{
    if(munmap(ptr, sz) == -1)
        fatal("can't munmap chunk @ %p: %s.", ptr, strerror(errno));
}
```

To ensure the required alignment, two mechanisms are used. If the address returned by `mmap` is found to be misaligned, it's immediately unmapped, and a mapping of twice the size is requested. An aligned address is computed within this, and the surplus space before and after the aligned chunk is unmapped.

⟨*mmap_aligned_chunk*⟩≡
```
static chunk_t *mmap_aligned_chunk(void *hint)
{
    uint8_t *head, *ptr = mmap_chunk(hint, CHUNKSIZE);

    if((uintptr_t)ptr & CHUNKMASK)
    {
        // throw it back
        munmap_chunk(ptr, CHUNKSIZE);
```

```
        // get one twice the size
        head = mmap_chunk(hint, CHUNKSIZE<<1);
        // align within that
        ptr = (uint8_t *)(((uintptr_t)head + CHUNKMASK) & ~CHUNKMASK);
        // give back the head
        munmap_chunk(head, ptr - head);
        // and tail
        munmap_chunk(ptr + CHUNKSIZE,
                     (head + (CHUNKSIZE<<1)) - (ptr + CHUNKSIZE));
    }
    return (chunk_t *)ptr;
}
```

The next-highest aligned address is stored, to be passed as the `hint` argument to the next `mmap` call – ideally avoiding further misalignment.

All the `page_t` descriptors are initially linked into the chunk's free list, and the chunk is accounted for and linked into the global chunk list.

⟨*chunk_create*⟩≡

```
static void *next_mmap = NULL;
static chunk_t *chunk_create()
{
    chunk_t *chunk = mmap_aligned_chunk(next_mmap);

    next_mmap = (uint8_t *)chunk + CHUNKSIZE;
    list_init(&chunk->free_head);
    chunk->nlive = 0;
    for(page_t *page = chunk->pages; page < &chunk->pages[NPAGES]; page++)
    {
        list_init(&page->page_list);
        list_add_before(&chunk->free_head, &page->page_list);
    }
    list_init(&chunk->chunk_list);
    list_add(&chunk_head, &chunk->chunk_list);
    gc_nchunks++;
    gc_stats.nchunks++;
    return chunk;
}
```

As a chunk can only be freed when its pages are all empty, they don't need special treatment during deallocation.

⟨*chunk_free*⟩≡

```
static void chunk_free(chunk_t *chunk)
{
    list_remove(&chunk->chunk_list);
    munmap_chunk(chunk, CHUNKSIZE);
    gc_nchunks--;
}
```

When the allocator needs an empty page, one can be taken from a chunk by removing the first page from its free list.

⟨*chunk_get_page*⟩≡

```
static inline page_t *chunk_get_page(chunk_t *chunk)
{
    page_t *page = container_of(chunk->free_head.next, page_t, page_list);

    list_remove(&page->page_list);
    return page;
}
```

(This assumes that `!list_isempty(&chunk->free_head)`.)

25.1.2 Pages

A page is either empty, or subdivided into objects of identical size. In the latter case, a pool will record the object size in the `.objsz` field when adding the page.

An empty page is on its chunk's free list. Otherwise it's on its pool's page list. The `.page_list` link suffices for both cases.

`.markbits` points to an out-of-line buffer containing one bit per object on the page. Each bit is set when the corresponding object is marked; it's cleared after each collection.

`.marked` is set when any object on the page is been marked. It's examined during sweeping to quickly determine whether the page is empty.

The `.free` field points to the first unused object on the page. If this is `NULL`, the page has no free objects left.

⟨*page_t*⟩≡
```
typedef unsigned short objsz_t;
typedef struct
{
    list_t page_list;
    SLIST(free_obj_t) free;
    bitmap_t *markbits;
    objsz_t objsz;
    bool marked;
} page_t;
```

⟨*pages*⟩≡
```
⟨page_storage⟩
⟨page_for_object⟩
⟨page_object_index⟩
⟨object_is_free⟩
⟨page_link_free_object⟩
⟨page_free_object⟩
⟨page_foreach_object⟩
⟨page_init⟩
⟨page_fini⟩
```

The address of some page's `chunk->pages[idx]` storage is located `(idx + NRESERVED) * PAGESIZE` bytes after the start of the chunk. Given a pointer to a `page_t`, both `chunk` and `idx` can be determined – since the `page_t` is contained in a chunk, its address is a valid argument to `chunk_for_addr`.

⟨*page_storage*⟩≡
```
static inline void *page_storage(page_t *page)
{
    chunk_t *chunk = chunk_for_addr(page);
    ptrdiff_t idx = page - chunk->pages;

    return (uint8_t *)chunk + ((idx + NRESERVED) << PAGEBITS);
}
```

Given a pointer `obj` to some object on a page, the appropriate `page_t` may be found *vice versa*.

⟨*page_for_object*⟩≡
```
static inline page_t *page_for_object(void *obj)
{
    chunk_t *chunk = chunk_for_addr(obj);
    unsigned idx = ((uint8_t *)obj - (uint8_t *)chunk) >> PAGEBITS;
```

```
    assert(idx >= NRESERVED);
    return &chunk->pages[idx-NRESERVED];
}
```

Mark bitmap manipulation will need the (zero-based) index of an object on its page (this assumes that `obj` is, in fact, on `page`).

⟨*page_object_index*⟩≡
```
static inline unsigned page_object_index(page_t *page, void *obj)
{
    uintptr_t offset = (uintptr_t)obj - ((uintptr_t)obj & ~PAGEMASK);
    assert(offset >= 0);
    return offset / page->objsz;
}
```

An object available for reuse is treated as a `free_obj_t`, and linked into its containing page's free list via its `.next` field. The list is singly linked since objects are only added or removed at the head.

⟨*free_obj_t*⟩≡
```
typedef struct free_obj
{
    robject_t hdr;
    SLIST(struct free_obj) next;
} free_obj_t;
```

Such objects have the sentinel value `r_type_freed` for their `.type`.

⟨*object_is_free*⟩≡
```
static const rtype_t *r_type_freed = (rtype_t *)0xDEADBEEF;
static inline bool object_is_free(free_obj_t *obj)
    { return obj->hdr.type == r_type_freed; }
```

Some objects hold pointers to unmanaged memory, and these will need to be freed before the object can be reused. The destructor `r_free` will invoke the appropriate callback for the object's type.

When debugging, storage is overwritten with a known bit pattern to aid in detecting object reuse bugs.

⟨*page_free_object*⟩≡
```
static inline void page_free_object(page_t *page, free_obj_t *obj)
{
    r_free(obj);
#ifndef NDEBUG
    memset(obj, 0xFE, page->objsz);
#endif
    page_link_free_object(page, obj);
}
```

Initialising a free object is a matter of setting its `.type` and linking it at the head of the page's free list.

When debugging with Valgrind (Nethercote and Seward, 2007), storage is also marked to provoke an error if inadvertantly accessed (the type pointer must stay accessible, though, since it's read during the sweep phase to determine whether or not an unmarked object is fresh garbage.)

⟨*page_link_free_object*⟩≡
```
static inline void page_link_free_object(page_t *page, free_obj_t *obj)
{
    obj->hdr.type = (rtype_t *)r_type_freed;
    slist_push(page->free, obj, next);
```

```
#ifdef HAS_VALGRIND
    VALGRIND_MAKE_MEM_NOACCESS((uint8_t *)obj + sizeof(rtype_t *),
                               page->objsz - sizeof(rtype_t *));
#endif
}
```

We use a convenient macro to iterate over all the objects on the page with descriptor `page_t *pg`. `obj` may be a pointer of any type.

⟨*page_foreach_object*⟩≡

```
#define page_foreach_object(pg, obj)                                   \
    for(void *_p = page_storage(pg);                                    \
        _p <= (void *)((uint8_t *)page_storage(pg)+PAGESIZE-pg->objsz)  \
            && (obj = _p);                                              \
        _p = (uint8_t *)_p + pg->objsz)
```

When a page is added to a pool, all its objects are initially added to its free list (and their `free_obj_t` fields marked accessible, when debugging under Valgrind.)

⟨*page_init*⟩≡

```
static page_t *page_init(page_t *page, objsz_t objsz, unsigned nobj)
{
    free_obj_t *obj;

    *page = (page_t) {
        .marked = false,
        .objsz = objsz,
        .markbits = bitmap_create(nobj),
        .free = NULL
    };
    page_foreach_object(page, obj)
    {
#ifdef HAS_VALGRIND
        VALGRIND_MAKE_MEM_DEFINED(obj, sizeof(free_obj_t));
#endif
        page_link_free_object(page, obj);
    }
    return page;
}
```

An empty page may be removed from its pool and placed on its chunk's free list (when debugging under Valgrind, its memory is marked inaccessible.)

⟨*page_fini*⟩≡

```
static void page_fini(page_t *page)
{
    chunk_t *chunk = chunk_for_addr(page);
    bitmap_free(page->markbits);
    list_add(&chunk->free_head, &page->page_list);
#ifdef HAS_VALGRIND
    VALGRIND_MAKE_MEM_NOACCESS(page_storage(page), PAGESIZE);
#endif
}
```

25.1.3 Pools

A pool satisfies allocation requests for a size class of objects.

⟨*pools*⟩≡

⟨*pool globals*⟩
⟨*pool_for_size*⟩

⟨*pool_add_page*⟩
⟨*pool_remove_page*⟩
⟨*pool_init*⟩
⟨*pool_fini*⟩

The list of `page_t`s in the `.page_head` field provides the storage from which to do so. The `.nobj` and `.objsz` fields record the number of objects per page and the size (in bytes) of the objects for which the pool is responsible.

The `.open` field is set to the last page from which an object was allocated, in the assumption it is not yet full.

The remaining fields are bookkeeping and may be of use to the heap sizing algorithm. `.npages` counts pages, the others count objects.

⟨*pool_t*⟩≡
```
typedef struct
{
    list_t page_head;
    page_t *open;
    objsz_t objsz;
    unsigned nobj;
    unsigned npages, nalloc, nfree, nswept;
} pool_t;
```

There is a `pool_t` for each size class of object, measured in natural machine words, up to the maximum required by the runtime.

There's a separate pool for types (Chapter 20) – the destructor callback invoked by `r_free` is a field of the type object. So the type must not be destroyed before the object, even if both became garbage during the same collection.

⟨*pool globals*⟩≡
```
#define MAX_WORDS 9
static const unsigned objwords[] = { 2, 3, 4, 5, 6, MAX_WORDS };
static pool_t *sizepools[MAX_WORDS];
#define NPOOLS ((unsigned) lengthof(objwords)+1)
static pool_t pools[NPOOLS], *type_pool = &pools[NPOOLS-1];
```

Dividing bytes to words and indexing into `sizepools` returns the pool from which to allocate an object of the desired size.

⟨*pool_for_size*⟩≡
```
static inline pool_t *pool_for_size(objsz_t sz)
    { return sizepools[sz >> WORDSHIFT]; }
```

A page added to a pool will always be empty, and it'll already have been removed from its chunk's list.

⟨*pool_add_page*⟩≡
```
static void pool_add_page(pool_t *pool, page_t *page)
{
    page_init(page, pool->objsz, pool->nobj);
    list_add(&pool->page_head, &page->page_list);
    pool->nfree += pool->nobj;
    pool->npages++;
}
```

Removing a page from a pool is a simple inverse. (Needless to say, this does not consider the case in which the page being removed contains live objects.)

⟨*pool_remove_page*⟩≡
```
static void pool_remove_page(pool_t *pool, page_t *page)
{
```

```
    assert(pool->open != page);
    pool->npages--;
    pool->nfree -= pool->nobj;
    list_remove(&page->page_list);
    page_fini(page);
}
```

Initialisation is straightforward...

⟨*pool_init*⟩≡

```
static void pool_init(objsz_t sz, pool_t *pool)
{
    *pool = (pool_t) {
        .page_head = LIST_INIT(pool->page_head),
        .objsz = sz,
        .nobj = PAGESIZE / sz,
    };
}
```

... as is destruction. All live objects on each page are destroyed. As the page is being removed from the list being traversed, `safe`ty is required.

⟨*pool_fini*⟩≡

```
static unsigned page_empty(page_t *page);
static void pool_fini(pool_t *pool)
{
    page_t *page, *tmp;

    list_foreach_entry_safe(&pool->page_head, page, tmp, page_list)
    {
        page_empty(page);
        list_remove(&page->page_list);
        page_fini(page);
    }
}
```

25.2 Allocation

Given a particular pool, if we can find a page with at least one free object, we can use it to satisfy an allocation request.

⟨*allocation*⟩≡
⟨*alloc_from_page*⟩
⟨*pool_get_page*⟩
⟨*alloc_from_pool*⟩
⟨*alloc_for_vec*⟩

Allocating from such a page is trivial – remove the object from the free list, and return. (Under Valgrind, the object's storage is marked as accessible.)

⟨*alloc_from_page*⟩≡

```
static inline void *alloc_from_page(page_t *page)
{
    free_obj_t *obj = page->free;

#ifdef HAS_VALGRIND
    VALGRIND_MAKE_MEM_DEFINED(obj, page->objsz);
#endif
    slist_remove_head(page->free, next);
    return obj;
}
```

The fast-path is taken when the pool has an `.open` page with a `.free` object. Otherwise we search the pool's pages for one; if none is found, we request an empty page from a chunk, and add it to the pool.

⟨*pool_get_page*⟩≡
```
static page_t *pool_get_page(pool_t *pool)
{
    page_t *page = pool->open;
    chunk_t *chunk;

    if(page && page->free)
        return page;
    list_foreach_entry(&pool->page_head, page, page_list)
        if(page->free)
            return page;
    list_foreach_entry(&chunk_head, chunk, chunk_list)
    {
        if(!list_isempty(&chunk->free_head))
        {
            page = chunk_get_page(chunk);
            pool_add_page(pool, page);
            return page;
        }
    }
    return NULL;
}
```

If there are none of those left, we have failed. `NULL` is returned to notify the caller, `alloc_from_pool`.

This makes up to two calls to `pool_get_page`. If the first call fails, a garbage collection is run and a second call is made. If this fails, a new chunk is requested, and an empty page from that is added to the pool.

A garbage collection is also run if the total amount of unmanaged memory `gc_vec_bytes` exceeds the `gc_vec_target` threshold. Otherwise, repeated allocation of large short-lived vectors could consume all available memory before running out of free objects and triggering a collection.

The pool's `.open` page and counters are updated, in any case, and the object returned.

⟨*alloc_from_pool*⟩≡
```
static unsigned gc_nchunks_target = 1;
static size_t gc_vec_target = 2*1024*1024;
static size_t gc_vec_bytes, gc_vec_swept;
static inline void *alloc_from_pool(pool_t *pool, size_t vecsz)
{
    page_t *page = pool_get_page(pool);
    bool obj_gc = (gc_nchunks >= gc_nchunks_target);
    bool vec_gc = (gc_vec_bytes + vecsz) > gc_vec_target;

#ifdef STRESS_GC
    if(true)
#else
    if(vec_gc || (obj_gc && !page))
#endif
    {
        if(gc_enabled)
        {
            gc_collect();
```

```
            page = pool_get_page(pool);
        }
        else
            gc_request = true;
    }
    if(!page)
    {
        page = chunk_get_page(chunk_create());
        pool_add_page(pool, page);
    }
    pool->open = page;
    pool->nalloc++;
    pool->nfree--;
    assert(pool->nfree < UINT_MAX);
    return alloc_from_page(page);
}
```

Unmanaged memory is allocated by the `libc`. `gc_alloc_vec` will call `alloc_for_vec` after `alloc_from_pool` if any unmanaged memory was requested. `gc_vec_target` is updated to prevent the `vec_gc` condition causing unnecessary collections during future allocations.

Note that the returned memory is not zeroed, so if it is to contain object pointers visible to the collector it must be initialised prior to the next allocation.

⟨*alloc_for_vec*⟩≡

```
static inline void *alloc_for_vec(size_t vecsz)
{
    if(vecsz == 0)
        return NULL;
    gc_vec_bytes += vecsz;
    gc_vec_target = max(gc_vec_target, gc_vec_bytes);
    gc_stats.vec_bytes += vecsz;
    return xmalloc(vecsz);
}
```

25.3 Collection

The garbage collector is invoked when a free object can't be found to satisfy an allocation, or when the amount of unmanaged memory allocated grows beyond a threshold value.

⟨*collection*⟩≡
⟨*queue*⟩
⟨*roots*⟩
⟨*marking*⟩
⟨*sweeping*⟩
⟨*sizing*⟩
⟨*gc_collect*⟩

Collection proceeds in phases. A set of preregistered *root* objects are *marked*, then marked objects are *scanned* in a loop. When an object is scanned, objects referenced by internal pointers are marked. When no more objects remain to be scanned, the collection is complete. Unmarked objects are considered garbage and reclaimed for later reuse.

Empty pages are released to their chunks. Chunks surplus to requirements (and with all pages empty) are released to the operating system. The unmanaged memory threshold is also updated.

⟨*gc_collect*⟩≡
```
void gc_collect()
{
    assert(gc_enabled);
    gc_request = false;
    collect_begin();
    collect_mark();
    collect_sweep();
    collect_resize();
    gc_stats.ncollect++;
}
```

25.3.1 Scan Queue

The scan queue contains objects which have been marked but not yet scanned.

⟨*queue*⟩≡
  ⟨*qfrag_t*⟩
  ⟨*queue globals*⟩
  ⟨*advance*⟩
  ⟨*qfrag_init*⟩

It comprises a singly-linked list of queue fragments, headed by the statically allocated fragment `qhead`; the following fragments, if any, are `malloc`ed. `qtail` points to the last one, or `NULL` if scanning hasn't started yet.

⟨*queue globals*⟩≡
```
static qfrag_t qhead, *qtail;
```

A queue fragment is `qfrag_t` structure, containing a ring buffer holding a constant number of object pointers, two indices, and a pointer to the next fragment (or `NULL`.)

⟨*qfrag_t*⟩≡
```
#define QFRAG_LEN 256
#define QFRAG_MASK (QFRAG_LEN-1)
typedef struct qfrag
{
    robject_t *objs[QFRAG_LEN];
    unsigned s, m;
    SLIST(struct qfrag) next;
} qfrag_t;
```

⟨*qfrag_init*⟩≡
```
static inline void qfrag_init(qfrag_t *frag)
{
    *frag = (qfrag_t) {
        .s = 0,
        .m = 0,
        .next = NULL
    };
}
```

`.s` is the scan index and `.m` the mark index (into the `.objs` buffer.) Each is advanced modulo `QFRAG_LEN` as scanning proceeds and as more objects are added to the queue, respectively.

⟨*advance*⟩≡
```
static inline unsigned advance(unsigned i)
    { return (i + 1) & QFRAG_MASK; }
```

25.3.2 Roots

A module that holds references to objects across a collection must register one or more `gc_root_t`s. The callback `.fn` should then invoke `gc_mark` on the objects that ought to be considered live.

(It's a `struct` so that, when embedded in a larger context structure, the callback can use `container_of` to compute the address of the latter from the former.)

⟨*gc_root_t*⟩≡
```
typedef struct gc_root_s gc_root_t;
typedef void (*gc_root_fn)(gc_root_t *);
typedef struct gc_root_s
{
    list_t roots_list;
    gc_root_fn fn;
} gc_root_t;
```

Registered roots are linked into a global list.

⟨*roots*⟩≡
```
static list_t roots_head = LIST_INIT(roots_head);
void gc_register(gc_root_t *root)
{
    list_init(&root->roots_list);
    list_add(&roots_head, &root->roots_list);
}
void gc_unregister(gc_root_t *root)
{
    list_remove(&root->roots_list);
}
⟨collect_begin⟩
```

To start a collection, the scan queue is initialised, and the registered callbacks invoked to populate the root set.

⟨*collect_begin*⟩≡
```
static void collect_begin()
{
    gc_root_t *root;

    qfrag_init(&qhead);
    qtail = NULL;
    list_foreach_entry(&roots_head, root, roots_list)
        root->fn(root);
}
```

25.3.3 Marking

A marked object has been reached from the root set by following pointers, so can't be reclaimed as garbage. In Dijkstra et al. (1978)'s tri-colour abstraction, a marked object is *grey* if it's on the queue and *black* if not.

⟨*marking*⟩≡
```
⟨mark_one_object⟩
⟨enqueue_one_object⟩
⟨gc_mark⟩
⟨collect_mark⟩
```

When an object is added to the current queue fragment, the mark index is advanced. if it 'catches up' to the scan index, the fragment is full and a new one must be allocated.

This is detected when `.m` is equal to `.s` and it's not the first object marked in the collection (since both indices are equal to `0`, in that case.)

The intuition is, in 'small' (narrow or shallow) parts of the object graph, marking doesn't get much ahead of scanning, so a smallish buffer will suffice. But burst capacity must be available, for parts of the graph which aren't.

⟨*enqueue_one_object*⟩≡
```
static inline void enqueue_one_object(robject_t *obj)
{
    if(!qtail)
        qtail = &qhead;
    else if(qtail->m == qtail->s)
    {
        qfrag_t *next = xmalloc(sizeof(*next));

        qfrag_init(next);
        qtail->next = next;
        qtail = next;
    }
    qtail->objs[qtail->m] = obj;
    qtail->m = advance(qtail->m);
}
```

An object is marked when the corresponding bit in its page's `.markbits` bitmap is set. To (potentially) speed sweeping, the page itself is also marked. The previous mark state is returned.

⟨*mark_one_object*⟩≡
```
static inline bool mark_one_object(robject_t *obj)
{
    page_t *page = page_for_object(obj);
    unsigned idx = page_object_index(page, obj);

    if(!bitmap_get_bit(page->markbits, idx))
    {
        page->marked = true;
        bitmap_set_bit(page->markbits, idx);
        return true;
    }
    return false;
}
```

`gc_mark` is the entry point for the runtime marking of live objects. If the given pointer is non-`NULL` and hasn't already been marked in this collection, it is added to the scan queue.

⟨*gc_mark*⟩≡
```
void gc_mark(void *ptr)
{
    if(ptr && mark_one_object(ptr))
        enqueue_one_object(ptr);
}
```

With the root set marked, marking (advancing `scan.m` and creating new fragments) and scanning (advancing `scan.s` and freeing completed ones) are interleaved until the queue is emptied.

⟨*collect_mark*⟩≡

```
static void collect_mark()
{
    for(qfrag_t *next, *scan = &qhead; scan; scan = next)
    {
        do
        {
            r_gc(scan->objs[scan->s]);
            scan->s = advance(scan->s);
        }
        while(scan->s != scan->m);
        next = scan->next;
        if(scan != &qhead)
            xfree(scan);
    }
}
```

25.3.4 Sweeping

The sweep phase links unmarked objects back into their pages' free lists, for later reuse.

⟨*sweeping*⟩≡
  ⟨*page_sweep*⟩
  ⟨*page_empty*⟩
  ⟨*pool_sweep*⟩
  ⟨*collect_sweep*⟩

Sweeping a page is a matter of iterating over the contained objects, freeing those which are unmarked and not already known to be free (i.e. that became garbage during this collection). The mark bitmap and flag are reset, and the count of objects swept is returned.

⟨*page_sweep*⟩≡

```
static unsigned page_sweep(page_t *page)
{
    free_obj_t *obj;
    unsigned idx = 0, nswept = 0;
    chunk_t *chunk = chunk_for_addr(page);

    page_foreach_object(page, obj)
    {
        if(!bitmap_get_bit(page->markbits, idx)
           && !object_is_free(obj))
        {
            page_free_object(page, obj);
            nswept++;
        }
        idx++;
    }
    bitmap_reset(page->markbits, idx);
    page->marked = false;
    chunk->nlive++;
    return nswept;
}
```

A page that has become empty will be returned to its chunk's free list. Linking its free objects isn't necessary; `page_empty` just runs the destructors of the contained objects.

⟨*page_empty*⟩≡

```
static unsigned page_empty(page_t *page)
{
```

```
    free_obj_t *obj;
    unsigned nswept = 0;

    page_foreach_object(page, obj)
    {
        if(!object_is_free(obj))
        {
            r_free(obj);
            nswept++;
        }
    }
    return nswept;
}
```

A page without its `.marked` flag set has no live objects, so is empty. Empty pages are promptly removed from the pool. Counters (and the `.open` page) are kept up-to-date. Otherwise the page's objects must be examined.

⟨*pool_sweep*⟩≡

```
static void pool_sweep(pool_t *pool)
{
    page_t *page, *tmp;
    unsigned nswept = 0; // mmm, fresh garbage.

    list_foreach_entry_safe(&pool->page_head, page, tmp, page_list)
    {
        if(!page->marked)
        {
            nswept += page_empty(page);
            if(pool->open == page)
                pool->open = NULL;
            pool_remove_page(pool, page);
        }
        else
            nswept += page_sweep(page);
    }
    pool->nalloc -= nswept;
    pool->nfree += nswept;
    pool->nswept = nswept;
}
```

Sweeping the heap just sweeps each pool in order (`type_pool` is the last element in `pools`, so objects are destroyed before their types.)

⟨*collect_sweep*⟩≡

```
static void collect_sweep()
{
    gc_vec_swept = 0;

    for(int i=0; i<NPOOLS; i++)
        pool_sweep(&pools[i]);
}
```

25.3.5 Sizing

As the user program executes, the way it uses memory can change – for example, during initialisation many long-lived objects might be allocated, but at other times most objects might become garbage relatively quickly.

The allocator keeps a target value for the number of allocated chunks and a threshold for unmanaged memory (i.e. vector bodies.) At the end of each collection, these are recomputed.

⟨*sizing*⟩≡
  ⟨*div_round_up*⟩
  ⟨*resize constants*⟩
  ⟨*target_size*⟩
  ⟨*target_vec_size*⟩
  ⟨*collect_resize*⟩

Rounding an integer division upwards slightly overestimates the result.

⟨*div_round_up*⟩≡
```
static inline intmax_t div_round_up(intmax_t a, int b)
    { return (a > 0) ? (a + b - 1) / b : a / b; }
```

⟨*resize constants*⟩≡
```
#define HEADROOM_DIV 4
#define SWEPT_DIV 2
```

We want to ensure that, for each pool, at least `(.nalloc / 4) + (.nswept / 2)` objects can be allocated before needing to run another collection. These should preferably come from pages the pool already owns (i.e. `.nfree`), or from unallocated pages in existing chunks. If these are insufficient, the target heap size is increased.

If there are more unallocated pages than desired, the heap might be too large. The target heap size is reduced, to allow some empty chunks (fragmentation permitting) to be freed.

⟨*target_size*⟩≡
```
static unsigned target_size(unsigned nchunks, unsigned target)
{
    chunk_t *chunk;
    page_t *page;
    intmax_t delta = 0;

    for(int i=0; i<NPOOLS; i++)
    {
        pool_t *pool = &pools[i];
        intmax_t ndelta = div_round_up(pool->nalloc, HEADROOM_DIV);

        ndelta += div_round_up(pool->nswept, SWEPT_DIV);
        ndelta -= pool->nfree;
        delta += max(0, ndelta) * (size_t)pool->objsz;
    }
    list_foreach_entry(&chunk_head, chunk, chunk_list)
        list_foreach_entry(&chunk->free_head, page, page_list)
            delta -= PAGESIZE;
    target = nchunks + div_round_up(delta, CHUNKSIZE);
    return max(1, target);
}
```

The unmanaged memory threshold is updated proportional to the current value, if it's not within 30% – 70% of the target. This is a very simple heuristic, similar to the one used in R (R Core Team, 2017, src/main/memory.c).

⟨*target_vec_size*⟩≡
```
static size_t target_vec_size(size_t bytes, size_t target)
{
    size_t delta = 0;
```

```
    if(bytes > target * 0.7)
        delta = (80 * 1024) + bytes * 0.1;
    else if(bytes < target * 0.3)
        delta = bytes * -0.2;
    return max(target + delta, 0);
}
```

As the final step in collection, the target sizes are recomputed and the chunk list traversed. Empty chunks which are surplus to the new maximum are freed. The list is reordered (via a temporary) – chunks with free pages sorting before those without (so that the loop over chunks in `pool_get_page` terminates sooner.)

⟨*collect_resize*⟩≡

```
static void collect_resize()
{
    list_t tmp_head = LIST_INIT(tmp_head);
    chunk_t *chunk, *tmp;

    gc_nchunks_target = target_size(gc_nchunks, gc_nchunks_target);
    gc_vec_target = target_vec_size(gc_vec_bytes, gc_vec_target);
    list_foreach_entry_safe(&chunk_head, chunk, tmp, chunk_list)
    {
        if(chunk->nlive == 0 && gc_nchunks > gc_nchunks_target)
        {
            chunk_free(chunk);
        }
        else
        {
            list_remove(&chunk->chunk_list);
            if(chunk->nlive == NPAGES)
                list_add_before(&tmp_head, &chunk->chunk_list);
            else
                list_add(&tmp_head, &chunk->chunk_list);
            chunk->nlive = 0;
        }
    }
    assert(list_isempty(&chunk_head));
    list_splice_after(&chunk_head, &tmp_head);
}
```

Note that a single live object suffices to keep a chunk from being empty. The inflation of process working set this induces has not been an issue so far, but might be detrimental to longer-running workloads.

25.4 Entry Points

⟨*interface*⟩≡
⟨*gc_alloc_vec*⟩
⟨*gc_free_vec*⟩
⟨*gc_release*⟩
⟨*gc_set_enabled*⟩
⟨*gc_was_requested*⟩
⟨*gc_collect_stats*⟩
⟨*gc_init*⟩
⟨*gc_fini*⟩

The main entry point to this module is the gc_alloc_vec function. It allocates and returns an object of the given size sz and type type.

Note that the latter may be NULL during runtime bootstrap, when r_type_type hasn't yet been constructed.

⟨gc_alloc_vec⟩≡
```
void *gc_alloc_vec(rtype_t *type, size_t sz, size_t vecsz, void **pvec)
{
    robject_t *obj;
    pool_t *pool;

    if(!type || type == r_type_type)
        pool = type_pool;
    else
        pool = pool_for_size(sz);

    obj = alloc_from_pool(pool, vecsz); // may collect
    obj->type = type;
    gc_stats.obj_bytes += sz;

    if(pvec)
      *pvec = alloc_for_vec(vecsz);
    return obj;
}
```

If the caller requires some unmanaged memory (e.g. for a vector or array body stored out-of-line,) it will pass a non-NULL pvec; this will be set, on return, to an allocation of vecsz bytes. It may be freed (usually in the object's type's .ops.free function) by calling gc_free_vec.

⟨gc_free_vec⟩≡
```
void gc_free_vec(void *ptr, size_t sz)
{
    xfree(ptr);
    gc_vec_bytes -= sz;
    gc_stats.vec_bytes -= sz;
    gc_vec_swept += sz;
}
```

Most callers will invoke the convenience function gc_alloc, since most objects don't require extra memory.

⟨gc_alloc⟩≡
```
static inline void *gc_alloc(rtype_t *type, size_t sz)
{
    return gc_alloc_vec(type, sz, 0, NULL);
}
```

Some global counters keep track of collector events and bytes allocated.

⟨gc_stats_t⟩≡
```
typedef struct
{
    unsigned ncollect;
    unsigned nchunks;
    size_t obj_bytes;
    size_t vec_bytes;
} gc_stats_t;
```

⟨gc_stats⟩≡
```
static gc_stats_t gc_stats;
```

These are returned and zeroed with `gc_collect_stats`.

⟨*gc_collect_stats*⟩≡
```
void gc_collect_stats(gc_stats_t *stats)
{
    *stats = gc_stats;
    gc_stats = (gc_stats_t) { 0 };
}
```

To simplify implementation, the compiler runs with garbage collection disabled.

⟨*interface globals*⟩≡
```
static bool gc_enabled;
static bool gc_request;
```

The collection that can be triggered by allocation is disabled with `gc_set_enabled`.

⟨*gc_set_enabled*⟩≡
```
void gc_set_enabled(bool enable)
{
    gc_enabled = enable;
    if(!enable)
        gc_request = false;
}
```

Instead of running a collection, the allocator will set the `gc_request` flag will instead. This may be inspected with `gc_was_requested`.

⟨*gc_was_requested*⟩≡
```
bool gc_was_requested()
{
    return gc_request;
}
```

If the compiler determines an object was allocated unnecessarily, it can explicitly release its memory by calling `gc_release`. This doesn't modify the pool's `.open` page or the chunk's free list, so `gc_collect` should be called at the earliest opportunity.

⟨*gc_release*⟩≡
```
void gc_release(void *ptr)
{
    if(ptr)
    {
        page_t *page = page_for_object(ptr);
        pool_t *pool = pool_for_size(page->objsz);
        unsigned idx = page_object_index(page, ptr);
        assert(!object_is_free(ptr));
        bitmap_clear_bit(page->markbits, idx);
        page_free_object(page, ptr);
        pool->nalloc--;
        pool->nfree++;
    }
}
```

`gc_init` initialises the managed heap. Each pool is initialised with `pool_init`, and the `sizepools` array populated. Collection is initially disabled.

⟨*gc_init*⟩≡
```
void gc_init()
{
    list_init(&chunk_head);
    for(int i=0; i<NPOOLS-1; i++)
    {
```

```
        objsz_t sz = objwords[i] << WORDSHIFT;

        pool_init(sz, &pools[i]);
    }
    pool_init(sizeof(rtype_t), type_pool);
    for(int i=1, j=0; i<MAX_WORDS; i++)
    {
        if(i > objwords[j])
            j++;
        sizepools[i] = &pools[j];
    }
    gc_enabled = false;
}
```

`gc_fini` tears down the heap structures and releases mapped memory to the system.

⟨*gc_fini*⟩≡

```
void gc_fini()
{
    chunk_t *chunk, *tmp;

    for(int i=0; i<NPOOLS-1; i++)
        pool_fini(&pools[i]);
    pool_fini(type_pool);

    list_foreach_entry_safe(&chunk_head, chunk, tmp, chunk_list)
        chunk_free(chunk);
}
```

Miscellanea

⟨*includes*⟩≡

```
#define HAS_VALGRIND
//#define STRESS_GC
#include "global.h"
#include <errno.h>
#include <sys/mman.h>
#ifdef HAS_VALGRIND
#include <valgrind/memcheck.h>
#endif
```

⟨*rt/gc.h*⟩≡

```
⟨gc_root_t⟩
⟨gc_stats_t⟩
⟨prototypes⟩
⟨gc_alloc⟩
```

⟨*prototypes*⟩≡
```
void gc_register(gc_root_t *root);
void gc_unregister(gc_root_t *root);
void gc_collect();
void gc_mark(void *ptr);
void gc_collect_stats(gc_stats_t *stats);
void *gc_alloc_vec(rtype_t *type, size_t sz, size_t vecsz, void **pvec);
void gc_free_vec(void *ptr, size_t vecsz);
void gc_release(void *ptr);
void gc_set_enabled(bool enable);
bool gc_was_requested();
void gc_init();
void gc_fini();
```

Chapter 26

Virtual Machine

The code[1] generated by the compiler is executed by a *virtual machine* which is *directly threaded* – each instruction is interpreted as the address of the routine that provides its implementation (Ertl and Gregg, 2003). This design was chosen to maximise performance while maintaining simplicity.

⟨*vm/vm.c*⟩≡
  ⟨*includes*⟩
  ⟨*context*⟩
  ⟨*vm macros*⟩
  ⟨*instruction macros*⟩
  ⟨*op_width_t*⟩
  ⟨*op_type_t*⟩
  ⟨*universal call*⟩
  ⟨*populate_closure*⟩
  ⟨*init_stack*⟩
  ⟨*vm_execute*⟩

26.1 Stack Frame

The primary addressable storage for the VM is the *stack*. This is subdivided into *frames*, each of which holds a saved state record, and the arguments and values local to the currently executing function.

Values on the stack are addressed by signed offset from the base of the current frame. This scheme is similar to that described in Ierusalimschy et al. (2005).

← low addresses

`.argsz`	`vm_act_rec_t`		`(.loc_scalsz)`	`.loc_sz`	
call frame	activation record	`bp`	local scalars	pointers	`sp`

The stack frame is divided into three areas. When a function is invoked, a *call frame* is cleared on the stack. This begins with an `argbits_t` value, with a bit set for each supplied argument, followed by the values of the actual arguments at the `.offsets` specified by the called function's signature. Following this is an *activation record* in which is saved the calling function's execution state. Next is the called function's *local area*, holding temporary values, which is segregated by type – first scalars, then references. The former is `.loc_scalsz` bytes in size; the local area is `.loc_sz` bytes in total. The VM's *base pointer* addresses the first byte in the local area; the *stack*

[1]bytecode, until a very recent change from indirect to direct threading.

pointer, one beyond the last.

⟨*vm_act_rec_t*⟩≡
```
typedef struct
{
    void *sp, *fp, *ip, *bp;
} vm_act_rec_t;
```

`init_stack` sets up the `vm`'s `.stack` to execute the closure given by `fp`. A top level closure takes no arguments, and an activation record with its `.fp` field set to `NULL` marks the initial stack frame, and the closure may have locals – so all these are zeroed. The initial value for the VM's base pointer is returned.

⟨*init_stack*⟩≡
```
static void *init_stack(vm_ctx_t *vm, rclosure_t *fp)
{
    size_t sz = sizeof(argbits_t) + sizeof(vm_act_rec_t);
    uint8_t *bp = (uint8_t *)vm->stack + sz;

    sz += fp->fn->loc_sz;
    memset(vm->stack, 0, sz);
    return bp;
}
```

26.2 Context

⟨*context*⟩≡
⟨*vm_error*⟩
⟨*vm_type_error*⟩
⟨*mark_locals*⟩
⟨*mark_args*⟩
⟨*vm_mark_roots*⟩
⟨*vm_update_ctx*⟩
⟨*vm_retain*⟩
⟨*vm_release*⟩
⟨*vm_init_ctx*⟩
⟨*vm_fini_ctx*⟩

The `vm_ctx_t` structure stores VM state shared with other subsystems. In the presence of appropriate mutual exclusion, multiple threads could execute code concurrently, each with their own context.

The `.stack` field points to a memory area `.stacksz` bytes in size. When execution succeeds, `.ret_val` will hold the value returned by the function; when it fails, `.err_msg` will hold a message indicating the reason. The `.err_buf` field is the `libc` jump buffer that backs the error handling mechanism.

The embedded `.gc_root` structure is registered with the garbage collector (Section 25.3). The `.retained` object array, current function `.fp`, and current stack frame base `.bp` supply the callback with the information it requires to mark live objects when a collection occurs during execution.

⟨*vm_ctx_t*⟩≡
```
typedef struct vm_ctx
{
    uint8_t *stack;
    size_t stacksz;
    robject_t *ret_val;
    char *err_msg;
    sigjmp_buf err_buf;
```

```
    gc_root_t gc_root;
    robject_array_t retained;
    rcallable_t *fp;
    uint8_t *bp;
} vm_ctx_t;
```

A subsystem intending to invoke the VM first calls `vm_init_ctx`, supplying the `vm` context, the initial `stack`, and its size `stacksz`. The context is initialised, and its `.gc_root` registered with the garbage collector.

⟨*vm_init_ctx*⟩≡
```
#define INIT_RETAINED 16
void vm_init_ctx(vm_ctx_t *vm, uint8_t *stack, size_t stacksz)
{
    *vm = (vm_ctx_t) {
        .stack = stack,
        .stacksz = stacksz,
        .gc_root = { .fn = vm_mark_roots },
        .retained = ARRAY_INIT,
        .err_msg = NULL,
        .fp = NULL
    };
    array_init(&vm->retained, INIT_RETAINED);
    gc_register(&vm->gc_root);
}
```

Once finished, `vm_fini_ctx` unregisters the `.gc_root`.

⟨*vm_fini_ctx*⟩≡
```
void vm_fini_ctx(vm_ctx_t *vm)
{
    array_fini(&vm->retained);
    gc_unregister(&vm->gc_root);
}
```

When an error occurs and execution cannot safely continue, `vm_error` will be called to format an informative message mentioning `src`, then exit via longjump to the error handler. The message is returned to the calling subsystem in the `.err_msg` field of the `vm` context.

⟨*vm_error*⟩≡
```
void vm_error(vm_ctx_t *vm, const char *src, const char *fmt, ...)
{
    va_list va;
    char *tmp;

    va_start(va, fmt);
    vasprintf(&tmp, fmt, va);
    va_end(va);
    asprintf(&vm->err_msg, "%s: %s", src, tmp);
    xfree(tmp);
    siglongjmp(vm->err_buf, -1);
}
```

Often, such an error is caused by an object of type `vtyp` being encountered by an instruction `src` where some `value` of an incompatible type `typ` is expected. The `vm_type_error` helper function is useful in these cases.

⟨*vm_type_error*⟩≡
```
static void vm_type_error(vm_ctx_t *vm, const char *src, const char *val,
                          rtype_t *typ, rtype_t *vtyp)
{
    vm_error(vm, src, "%s of type ‘%s‘ found where type ‘%s‘ expected.",
             val, rtype_name(vtyp), rtype_name(typ));
}
```

Builtin functions that allocate objects may need to protect them from inadvertant collection. If an object isn't transitively reachable from another root, it can be added to the `vm`'s context with `vm_retain`.

⟨*vm_retain*⟩≡
```
void *vm_retain(vm_ctx_t *vm, void *obj)
{
    array_push(&vm->retained, obj);
    return obj;
}
```

The last `n` objects retained can be released with `vm_release`.

⟨*vm_release*⟩≡
```
void vm_release(vm_ctx_t *vm, int n)
{
    array_drop(&vm->retained, n);
}
```

When the VM begins execution of a different function through a `call` or `ret`, it calls `vm_update_ctx` to update the stored function and base pointers.

⟨*vm_update_ctx*⟩≡
```
static inline void
vm_update_ctx(vm_ctx_t *vm, rcallable_t *fp, uint8_t *bp)
{
    vm->fp = fp;
    vm->bp = bp;
}
```

Along with the contents of the `.retained` array, these pointers are used by `vm_mark_roots` when called by the garbage collector, supplying initial values for `fp` and `bp` respectively.

`fp` is `NULL` when no function is being executed – possibly because it returned a result value. In this case, `.ret_val` is marked, but nothing else need be. Otherwise, the `.retained` objects are marked then the stack traversed, beginning at `bp`.

The function `fp` is `gc_mark`ed, as are its frame's local pointers (when it's not a builtin) and arguments. Then `bp` and `fp` are updated from the saved registers in the activation `record` below `bp`, and the loop continues down the stack, until all frames have been visited.

⟨*vm_mark_roots*⟩≡
```
static void vm_mark_roots(gc_root_t *root)
{
    vm_ctx_t *vm = container_of(root, vm_ctx_t, gc_root);
    rcallable_t *fp = vm->fp;
    uint8_t *bp = vm->bp;
    robject_t *obj;

    if(!vm->fp)
```

```
    {
        gc_mark(vm->ret_val);
        return;
    }
    array_foreach_entry(&vm->retained, obj)
        gc_mark(obj);
    while(bp)
    {
        vm_act_rec_t *rec = (vm_act_rec_t *)bp - 1;

        gc_mark(fp);
        if(!rcall_is_builtin(fp))
            mark_locals(bp, ((rclosure_t *)fp)->fn);
        mark_args(rec, fp);
        bp = rec->bp;
        fp = rec->fp;
    }
}
```

The local area starts at `bp` and is `.loc_sz` bytes in length. `mark_locals` ignores the initial `.loc_scalsz` bytes of scalars, and marks each object reference in the region that follows.

⟨*mark_locals*⟩≡

```
static void mark_locals(uint8_t *bp, rfunction_t *fn)
{
    for(robject_t **obj = (robject_t **)(bp + fn->loc_scalsz);
        obj < (robject_t **)(bp + fn->loc_sz); obj++)
        gc_mark(*obj);
}
```

A function's arguments are specified by its `sig`nature and are laid out in the call frame beginning `.argsz` bytes before the activation `record`. `mark_args` marks the value of each argument with reference type.

⟨*mark_args*⟩≡

```
static void mark_args(vm_act_rec_t *rec, rcallable_t *fp)
{
    funsig_t *sig = rcall_sig(fp);
    uint8_t *argbase = (uint8_t *)rec - sig->argsz;

    for(int i=0; i < sig->nargs; i++)
    {
        argdesc_t *arg = &sig->args[i];

        if(!rtype_is_scalar(arg->type))
            gc_mark(*(robject_t **)(argbase + arg->offset));
    }
}
```

26.3 Execution

Each instruction in the interpreted code comprises one `vm_instr_t` followed by zero or more operands, typically `op_stack_t`s or `op_offset_t`s.

⟨*vm_instr_t*⟩≡

```
typedef const void *vm_instr_t;
```

The `vm_execute` function interprets the code of the given `closure` within an initialised `vm` context. It returns `0` when successful, `1` in case of error; setting the context's `.ret_val` or `.err_msg` field, respectively.

Four local "registers" hold the VM's execution state, are and saved in the activation record pushed on the stack when a function is `call`ed: `ip` points to the next instruction, `fp` to the closure of the currently executing function, `bp` to the base of the stack frame, and `sp` to the top of the stack.

`callee` and `framesize` are used by the `call` instructions; `pret` points to the returned value transferred between `ret` and `complete`.

After the registers are initialised (and `init_stack` clears the call frame,) the error handler is set up with `sigsetjmp`. The `vm` context is updated, and the first instruction `DISPATCH`ed.

When `vm_error` is invoked, `sigsetjmp` returns a non-zero value. Execution is aborted, the context invalidated, and an indication of failure is returned to the caller.

⟨*vm_execute*⟩≡

```
const void *const *vm_instr;
int vm_execute(vm_ctx_t *vm, rclosure_t *closure)
{
    ⟨instruction table⟩
    op_code_t *ip = closure->fn->code;
    rclosure_t *fp = closure;
    uint8_t *bp = init_stack(vm, closure);
    uint8_t *sp = bp + closure->fn->loc_sz;

    rcallable_t *callee;
    op_offset_t framesize;
    void *pret;

    if(!sigsetjmp(vm->err_buf, 1))
    {
        vm_update_ctx(vm, &fp->base, bp);
        DISPATCH();
    }
    vm->ret_val = NULL;
    vm->fp = NULL;
    return 1;
    ⟨instructions⟩
}
```

Within the body of `vm_execute`, each instruction consists of a labelled block. The addresses of these labels are stored in the local `instr` array, indexed by opcode. This array forms a jump table for the direct-threaded bytecode, and is initialised with the `OPS_ALL` macro, each constituent `OP_DEF` expanding to an element initialiser.

When `vm_execute` is called with a `NULL` context during runtime initialisation (Section 19.6), it makes this table available to the code generation subsystem as `vm_instr`.

⟨*instruction table*⟩≡

```
#define OP_DEF(def...) [ENUM(def)] = &&LABEL(def)
static const void *const instr[] =
    {
        OPS_ALL
    };
#undef OP_DEF
if(!vm)
{
    vm_instr = instr;
```

```
    return 0;
}
```

A set of macros simplify instruction implementation. `ADVANCE` yields the value of `ip` and advances it by `n` bytes. `FETCH` retrieves the next `ctype` from the instruction stream. `DISPATCH` executes the next instruction. `SFETCH` and `OFETCH` are shorthand for fetching operands of types `op_stack_t` and `op_offset_t`. `STACK` retrieves the `ctype` at the offset `ofs` from `bp`.

Note that these values are not aligned – which has minimal performance impact on x86-64 hardware, but may be inefficient or illegal on other architectures.

⟨*vm macros*⟩≡

```
#define ADVANCE(n) ((ip += (n)) - (n))
#define FETCH(ctype) (*((ctype *)ADVANCE(sizeof(ctype))))
#define DISPATCH() goto *FETCH(vm_instr_t *);
#define SFETCH() FETCH(op_stack_t)
#define OFETCH() FETCH(op_offset_t)
#define STACK(ctype, ofs) *((ctype *)(bp + ofs))
```

In the instruction descriptions that follow, an `op_stack_t` operand is printed *without* brackets, an `op_offset_t` operand with [*square*] brackets, and a literal value with ⟨*angle*⟩ brackets.

26.4 Generic Instructions

26.4.1 Universal Call

call_uni *target*

Invokes the *target* `callable` object, with actual arguments specified by the following sub-ops in the instruction stream. These are responsible for forming the call frame expected by the `callee` – matching named arguments, unboxing or converting scalars, creating a rest vector, etc. When the frame is complete, `call_common` transfers control to the `callee`.

⟨*instructions*⟩≡

```
do_call_uni:
{
    op_stack_t target = SFETCH();
    callee = STACK(rcallable_t *, target);
    rtype_t *cl_type = r_typeof(callee);

    if(!rtype_is_callable(cl_type))
        vm_type_error(vm, "call", "callee", r_type_callable, cl_type);
    framesize = cl_type->sig->argsz;
    if(sp + framesize >= vm->stack + vm->stacksz)
        vm_error(vm, "call", "stack overflow.");
    memset(sp, 0, framesize);
    ip = vm_call(vm, cl_type->sig, ip, fp, sp, bp);
    sp += framesize;
    goto call_common;
}
```

⟨*universal call*⟩≡
  ⟨*kwd_pair_t*⟩
  ⟨*vm_call_ctx_t*⟩
  ⟨*set_arg*⟩
  ⟨*fail_rest*⟩
  ⟨*append_rest*⟩
  ⟨*finish_rest*⟩
  ⟨*vm_call*⟩

Unmatched arguments are collected as pairs of `.name`s and `.val`ues before being copied into the rest vector (if one is requested by the callee).

⟨*kwd_pair_t*⟩≡
```
typedef struct
{
    rsymbol_t *name;
    robject_t *val;
} kwd_pair_t;
```

The `.base` structure embedded in the `vm_call_ctx_t` is used by the argument matcher functions (Subsection 22.3.2). The `.sp` field points to the start of the call frame under construction. The `.rp` field points to temporary storage for the next unmatched argument, and `.nrest` counts those collected so far. The `.has_names` flag is set if any actual arguments in the call are named. `.tos` points to the end of the stack, and stack overflow is signalled via the `.vm` context.

⟨*vm_call_ctx_t*⟩≡
```
typedef struct
{
    call_ctx_t base;
    uint8_t *sp;
    kwd_pair_t *rp;
    int nrest;
    bool has_names;
    uint8_t *tos;
    vm_ctx_t *vm;
} vm_call_ctx_t;
```

`call_end` terminates the sub-operation sequence. Each of the others expresses a call to the appropriate matcher.

⟨*call subcodes*⟩≡
```
enum { OP_call_end, OP_call_kwd, OP_call_pos,
       OP_call_rest, OP_call_omit };
```

The `vm_call` function forms, at `sp`, a call frame suitable for a function with signature `sig`. The sub-ops at `ip` are interpreted; referring to argument names from the constant pool of `fp`, and copying values from the stack frame at `bp`.

The assignment callback `.append_rest` is set to `fail_rest` if the signature does not specify that it `.has_rest`. Unmatched arguments will be collected (temporarily) on the stack beyond the call frame, starting at `.rp`. With the `ctx` initialised, a loop interprets sub-opcodes until the `end` of the sequence.

⟨*vm_call*⟩≡

```
static op_code_t *vm_call(vm_ctx_t *vm, funsig_t *sig, op_code_t *ip,
                          rclosure_t *fp, uint8_t *sp, uint8_t *bp)
{
    op_code_t subop;
    vm_call_ctx_t ctx = {
        .base = {
            .posbits = 0,
            .argbits = 0,
            .sig = sig,
            .append_rest = sig->has_rest ? append_rest : fail_rest,
            .set_arg = set_arg
        },
        .sp = sp,
        .rp = (kwd_pair_t *)(sp + sig->argsz),
        .tos = vm->stack + vm->stacksz,
        .nrest = 0,
        .vm = vm
    };

    while((subop = *ip++) != OP_call_end)
    {
        switch(subop)
        {
            ⟨subop switch⟩
        }
    }
    ⟨finish call⟩
}
```

call_kwd [*idx*] *src*

Invokes `call_match_kwd`, supplying a name from the constant pool at *idx* and a value from the stack at *src*.

⟨*subop switch*⟩≡

```
case OP_call_kwd:
{
    op_offset_t idx = OFETCH();
    op_stack_t src = SFETCH();

    rsymbol_t *name = (rsymbol_t *)fp->fn->consts[idx];
    robject_t *val = STACK(robject_t *, src);
    if(!call_match_kwd(&ctx.base, name, val))
        vm_error(vm, "call", "couldn’t pass named argument.");
    break;
}
```

call_pos *src*

Invokes `call_match_pos`, supplying the value on the stack at *src* as the next positional argument.

⟨*subop switch*⟩+≡
```
case OP_call_pos:
{
    op_stack_t src = SFETCH();
    robject_t *val = STACK(robject_t *, src);

    if(!call_match_pos(&ctx.base, val))
        vm_error(vm, "call", "couldn't pass positional argument.");
    break;
}
```

call_kwd *src*

Invokes `call_match_rest` to match the elements of the vector at *src* as actual arguments (if it's not a vector, or doesn't have `object` element type, an error is signalled.)

⟨*subop switch*⟩+≡
```
case OP_call_rest:
{
    op_stack_t src = SFETCH();
    rvector_t *rest = STACK(rvector_t *, src);

    if(!rest)
        break;
    if(!r_subtypep(r_typeof(rest), r_type_vec_object))
        vm_type_error(vm, "call", "rest list", r_type_vec_object,
                      r_typeof(rest));

    robject_t **vals = rvec_elts(rest);
    rsymbol_t **names = rvec_is_named(rest) ? rvec_elts(rest->names)
                      : NULL;
    int nvals = rvec_len(rest);
    if(!call_match_rest(&ctx.base, (void **)vals, names, nvals))
        vm_error(vm, "call", "couldn't pass rest list.");
    break;
}
```

call_omit

Marks the argument as omitted by invoking `call_match_omit`.

⟨*subop switch*⟩+≡
```
case OP_call_omit:
    if(!call_match_omit(&ctx.base))
        vm_error(vm, "call", "couldn't pass omitted argument.");
    break;
```

After the call frame is populated, the `.argbits` set are checked against the signature's `.reqbits` – if any required arguments were not supplied, an error is signalled. The `argbits` are written as the first argument in the call frame at `sp`. If a rest vector is needed, it's allocated and filled by `finish_rest`, with the object reference written as the last argument.

The address of the next instruction is returned, to be saved in the activation record pushed before control is transferred to the callee.

⟨*finish call*⟩≡
```
if(sig->reqbits != (ctx.base.argbits & sig->reqbits))
    vm_error(vm, "call", "required argument not supplied.");
*(argbits_t *)sp = ctx.base.argbits;
if(sig->has_rest && ctx.nrest > 0)
{
    uint8_t *argp = sp + sig->argsz - sizeof(rvector_t *);
    *(rvector_t **)argp = finish_rest(&ctx);
}
return ip;
```

`set_arg` examines the actual argument's `val`ue to determine how to it is to be stored into the call frame at the formal `arg`ment's `.offset`.

If the latter is of scalar type, the former must be unboxed (if of the same type) or converted (if of different scalar type). Otherwise, the reference can be copied if the former is a subtype of the latter. If types fail to match, an error is signalled.

⟨*set_arg*⟩≡
```
static bool set_arg(void *ptr, argdesc_t *arg, void *val)
{
    vm_call_ctx_t *ctx = ptr;
    uint8_t *dest = ctx->sp + arg->offset;
    rtype_t *typ = arg->type, *vtyp = r_typeof(val);

    if(rtype_is_scalar(typ))
    {
        if(vtyp == typ)
            r_unbox(dest, val);
        else if(rtype_is_scalar(vtyp))
            scalar_convert(dest, val, typ, vtyp);
        else
            vm_type_error(ctx->vm, "call", "argument",
                          typ, vtyp);
    }
    else
    {
        if(!r_subtypep(vtyp, typ))
            vm_type_error(ctx->vm, "call", "argument",
                          typ, vtyp);
        *(robject_t **)dest = val;
    }
    return true;
}
```

`fail_rest` signals an error when an argument fails to match and no rest vector is being collected.

⟨*fail_rest*⟩≡
```
static bool fail_rest(void *ptr, rsymbol_t *name, void *val)
{
    vm_call_ctx_t *ctx = ptr;

    if(name)
        vm_error(ctx->vm, "call", "unknown argument ‘%s‘.",
                 name->string);
    else
        vm_error(ctx->vm, "call", "too many arguments.");
    return false;
}
```

`append_rest` stores at `.rp` (and counts in `.nrest`) the unmatched argument with value `val` and an optional `name`.

⟨*append_rest*⟩≡
```
static bool append_rest(void *ptr, rsymbol_t *name, void *val)
{
    vm_call_ctx_t *ctx = ptr;

    if((uint8_t *)ctx->rp > ctx->tos)
        vm_error(ctx->vm, "call", "too many arguments; out of stack.");
    if(name)
        ctx->has_names = true;
    *(ctx->rp++) = (kwd_pair_t) {
        .name = name,
        .val = val
    };
    ctx->nrest++;
    return true;
}
```

The unmatched arguments collected have their values copied into a new `rest` vector by `finish_rest`. If the call `.has_names`, their names are copied as well.

Garbage collection could occur in the `rvec_create` call but, by construction, all values in the call frame and temporary storage (where they're not visible to the collector) have been copied from locals or arguments frame (where they are.)

⟨*finish_rest*⟩≡
```
static rvector_t *finish_rest(vm_call_ctx_t *ctx)
{
    rvector_t *rest = rvec_create(r_type_vec_object, ctx->nrest);
    robject_t **vals = rvec_elts(rest);
    kwd_pair_t *rp = ctx->rp - ctx->nrest;

    vm_retain(ctx->vm, rest);
    for(int i = 0; i < ctx->nrest; i++)
        vals[i] = rp[i].val;
    if(ctx->has_names)
    {
        rsymbol_t **names = rvec_elts(rvec_add_names(rest));

        for(int i=0; i<ctx->nrest; i++)
            names[i] = rp[i].name;
    }
    vm_release(ctx->vm, 1);
```

```
    return rest;
}
```

26.4.2 Call & Return

call_fast *target*

Invokes the *target* `callable` object. The compiler is responsible for ensuring the call frame below `sp` contains the actual arguments in a layout conforming to the `callee`'s expectations. `call_common` is again responsible for transfer of control.

⟨*instructions*⟩+≡
```
do_call_fast:
{
    op_stack_t target = SFETCH();
    callee = STACK(rcallable_t *, target);
    framesize = r_typeof(callee)->sig->argsz;
    goto call_common;
}
```

After either kind of `call`, an activation record is pushed to save the current register values. The saved stack pointer is adjusted so that when it's restored the call frame will be discarded. The stack and base pointers are advanced past the activation record, and the `callee` invoked.

⟨*instructions*⟩+≡
```
call_common:
{
    *(vm_act_rec_t *)sp = (vm_act_rec_t)
        {
            .sp = sp - framesize,
            .fp = fp,
            .ip = ip,
            .bp = bp
        };
    sp += sizeof(vm_act_rec_t);
    bp = sp;
    if(rcall_is_builtin(callee))
        goto call_builtin;
    goto call_vm;
}
```

When the `callee` is a closure, invocation is a matter of adjusting the VM registers. The function pointer is changed to point to the `callee`; the instruction pointer, to the start of its `.code` buffer. The stack pointer is advanced to the end of the locals area, which is then cleared. After updating the `vm` context, the first instruction in the callee is finally `DISPATCH`ed.

⟨*instructions*⟩+≡
```
call_vm:
{
    rclosure_t *cl = (rclosure_t *)callee;
    op_offset_t loc_sz = cl->fn->loc_sz;

    fp = cl;
    ip = cl->fn->code;
    sp += loc_sz;
    if(sp >= vm->stack + vm->stacksz)
        vm_error(vm, "call", "stack overflow.");
```

```
        memset(sp - loc_sz, 0, loc_sz);
        vm_update_ctx(vm, &fp->base, bp);
        DISPATCH();
    }
```

A builtin `callee` has no local area, but returns its value via the top of the stack at `sp`. The `vm` context is updated and the builtin function `.fn` invoked; `ret_common` proceeds to pop the activation record. This is inefficient – builtins don't affect the VM registers – but simplifies garbage collection, since both kinds of function have the same stack frame layout.

⟨*instructions*⟩+≡
```
    call_builtin:
    {
        rbuiltin_t *bi = (rbuiltin_t *)callee;

        if(sp >= vm->stack + vm->stacksz - sizeof(double))
            vm_error(vm, "call", "stack overflow.");
        pret = sp;
        vm_update_ctx(vm, callee, bp);
        bi->fn(vm, bi, bp - sizeof(vm_act_rec_t), pret);
        goto ret_common;
    }
```

ret *result*

Takes a pointer to the *result* value on the stack, then pops the activation record via `ret_common` to resume execution in the function's caller.

The returned pointer is used by the next `complete` instruction, which immediately follows the `call`.

⟨*instructions*⟩+≡
```
    do_ret:
    {
        op_stack_t result = SFETCH();
        pret = bp + result;
        goto ret_common;
    }
```

The VM registers are restored from the activation record immediately below `bp`, and the `vm` context updated.

If the saved function pointer is `NULL`, the function returning is the `closure` that was passed in to begin with. Execution complete, the return value is saved in the context, and `vm_execute` returns success.

Otherwise, `DISPATCH` proceeds to the instruction following previous `call` (which will be a `complete`, if the return value is used).

⟨*instructions*⟩+≡
```
    ret_common:
    {
        vm_act_rec_t *rec = ((vm_act_rec_t *)bp) - 1;
        sp = rec->sp;
        fp = rec->fp;
        ip = rec->ip;
        bp = rec->bp;
        vm_update_ctx(vm, &fp->base, bp);
        if(!fp)
        {
```

```
        vm->ret_val = *(robject_t **)pret;
        vm->fp = NULL;
        return 0;
    }
    DISPATCH();
}
```

complete_box *target dest*

Retrieves the value returned by a previous `call_uni` to the *target*. If it's an unboxed scalar, allocates a box to hold it; under the universal calling convention a call's result must be a reference object. The value is placed on the stack at *dest*.

⟨*instructions*⟩+≡
```
do_complete_box:
{
    op_stack_t target = SFETCH(), dest = SFETCH();
    rtype_t *cl_type = r_typeof(STACK(rcallable_t *, target));
    rtype_t *ret_type = cl_type->sig->ret_type;
    if(rtype_is_scalar(ret_type))
    {
        robject_t *box = r_box(ret_type, pret);
        STACK(robject_t *, dest) = box;
    }
    else
    {
        STACK(robject_t *, dest) = *(robject_t **)pret;
    }
    DISPATCH();
}
```

missing *dest src* [*idx*]

Tests whether the `argbits_t` on the stack at *src* has the bit at *idx* set, storing the resulting boolean at *dest*.

⟨*instructions*⟩+≡
```
do_missing:
{
    op_stack_t dest = SFETCH();
    op_stack_t src = SFETCH();
    op_offset_t idx = OFETCH();
    argbits_t bits = STACK(argbits_t, src);
    rboolean_t val = argbits_missing(bits, idx);
    STACK(rboolean_t, dest) = val;
    DISPATCH();
}
```

frame [*size*] ⟨*bits*⟩

Clears a call frame of the given *size*; writing the literal *bits* as the implicit first argument, and incrementing the stack pointer.

⟨*instructions*⟩+≡
```
do_frame:
{
    op_offset_t size = OFETCH();
    argbits_t bits = FETCH(argbits_t);
    if(sp + size >= vm->stack + vm->stacksz)
        vm_error(vm, "frame", "stack overflow.");
    memset(sp, 0, size);
    *((argbits_t *)sp) = bits;
    sp += size;
    DISPATCH();
}
```

26.4.3 Control Flow

jump [*addr*]

Unconditionally branches to the instruction located at *addr*.

⟨*instructions*⟩+≡
```
do_jump:
{
    op_offset_t addr = OFETCH();
    ip = fp->fn->code + addr;
    DISPATCH();
}
```

if *pred* [*addr*]

Examine the boolean on the stack at *pred*. When `true`, branch to the instruction located at *addr*, instead of continuing with the next instruction. If the boolean is `NA`, an error is signalled.

⟨*instructions*⟩+≡
```
do_if:
{
    op_stack_t pred = SFETCH();
    op_offset_t addr = OFETCH();
    rboolean_t val = STACK(rboolean_t, pred);
    if(val == true)
        ip = fp->fn->code + addr;
    else if(val == rboolean_na)
        vm_error(vm, "if", "NA value found where true/false expected.");
    DISPATCH();
}
```

26.4.4 Lambda

lambda *dest* [*idx*]

Fetches the `rfunction_t` from the constant pool at index *idx*, and creates a new closure. The following sequence of sub-operations initialise its captured values. Stores the closure into the stack at *dest*.

⟨*instructions*⟩+≡
```
do_lambda:
{
    op_stack_t dest = SFETCH();
    op_offset_t idx = OFETCH();
    rfunction_t *fn = (rfunction_t *)fp->fn->consts[idx];
    rclosure_t *value = rcall_closure_create(fn->cl_type, fn);

    ip = populate_closure(value->env, ip, bp);
    STACK(rclosure_t *, dest) = value;
    DISPATCH();
}
```

`lambda_end` terminates the sequence; the other sub-ops copy values of particular widths.

⟨*lambda subcodes*⟩≡
```
enum { OP_lambda_end, OP_lambda_mov8,
       OP_lambda_mov32, OP_lambda_mov64 };
```

`populate_closure` reads sub-opcodes until the `end` is encountered. Each is followed by a `bp`-relative stack location; the specified value is copied into the closure's `env` buffer. The new value of `ip` is returned.

⟨*populate_closure*⟩≡
```
static op_code_t *populate_closure(void *env, op_code_t *ip, uint8_t *bp)
{
    op_code_t subop;
    uint8_t *ptr = env;

    while((subop = *ip++) != OP_lambda_end)
    {
        op_stack_t src = SFETCH();

        switch(subop)
        {
        case OP_lambda_mov8:
            *(uint8_t *)ptr = STACK(uint8_t, src);
            ptr += 1;
            break;
        case OP_lambda_mov32:
            *(uint32_t *)ptr = STACK(uint32_t, src);
            ptr += 4;
            break;
        case OP_lambda_mov64:
            *(uint64_t *)ptr = STACK(uint64_t, src);
            ptr += 8;
            break;
        }
    }
    return ip;
}
```

26.4.5 Global Variables

defvar [*name*] [*type*]

Creates a new global variable, with *name* and *type* found at the specified indices in the constant pool. This is always followed by a `setvar`.

⟨*instructions*⟩+≡
```
do_defvar:
{
    op_offset_t nidx = OFETCH(), tidx = OFETCH();
    rsymbol_t *sym = (rsymbol_t *)fp->fn->consts[nidx];
    rtype_t *type = (rtype_t *)fp->fn->consts[tidx];

    if(r_get_global(sym))
        vm_error(vm, "def", "can’t define ‘%s‘.", r_symstr(sym));
    r_create_global(sym, type, false);
    DISPATCH();
}
```

getvar_uni *dest* [*name*]

Looks up the global variable *name*d by the given constant, signalling an error if not found.

Boxes the variable’s value if it was declared with scalar type, otherwise just copies the reference. Either way, the value ends up on the stack at *dest*.

⟨*instructions*⟩+≡
```
do_getvar_uni:
{
    op_stack_t dest = SFETCH();
    op_offset_t idx = OFETCH();
    rsymbol_t *sym = (rsymbol_t *)fp->fn->consts[idx];
    rglobal_t *global = r_get_global(sym);

    if(!global)
        vm_error(vm, "get", "global ‘%s‘ not defined.", r_symstr(sym));
    if(global->decl && rtype_is_scalar(global->decl))
        STACK(robject_t *, dest) = r_box(global->decl, &global->val);
    else
        STACK(robject_t *, dest) = global->val.object;
    DISPATCH();
}
```

setvar_uni *value* [*name*]

Looks up the global variable with the constant *name*, creating it if not present.

If it's declared with a scalar type, unboxes the *value* from the stack into the `global`'s `.val` field, otherwise just copies the reference.

⟨*instructions*⟩+≡
```
do_setvar_uni:
{
    op_stack_t src = SFETCH();
    op_offset_t idx = OFETCH();
    rsymbol_t *sym = (rsymbol_t *)fp->fn->consts[idx];
    robject_t *obj = STACK(robject_t *, src);
    rglobal_t *global = r_get_global(sym);

    if(!global)
        global = r_create_global(sym, NULL, false);
    else if(global->is_const)
        vm_error(vm, "set", "invalid assignment to ‘const %s‘.",
                 r_symstr(sym));
    if(global->decl && rtype_is_scalar(global->decl))
    {
        rtype_t *type = r_typeof(obj), *decl = global->decl;
        void *ptr = &global->val;

        if(decl == type)
            r_unbox(ptr, obj);
        else if(rtype_is_scalar(type))
            scalar_convert(ptr, obj, decl, type);
        else
            vm_type_error(vm, "unbox", "value", decl, type);
    }
    else
        global->val.object = obj;
    DISPATCH();
}
```

26.4.6 General

const *dest* [*idx*]

Copies the value from index *idx* in the constant pool into the stack at *dest*. This is always a reference to an object.

⟨*instructions*⟩+≡
```
do_const:
{
    op_stack_t dest = SFETCH();
    op_offset_t idx = OFETCH();

    STACK(robject_t *, dest) = fp->fn->consts[idx];
    DISPATCH();
}
```

cellmake *dest* [*type*]

Creates a cell (Chapter 12) with the specified constant *type*, storing it to the stack at *dest*.

⟨*instructions*⟩+≡
```
do_cellmake:
{
    op_stack_t dest = SFETCH();
    op_offset_t idx = OFETCH();
    rtype_t *type = (rtype_t *)fp->fn->consts[idx];
    STACK(robject_t *, dest) = rcell_create(type);
    DISPATCH();
}
```

check *src* [*type*]

Signals an error if the reference object on the stack at *src* has a type which is not a subtype of the given constant *type*.

⟨*instructions*⟩+≡
```
do_check:
{
    op_stack_t src = SFETCH();
    op_offset_t idx = OFETCH();
    rtype_t *type = (rtype_t *)fp->fn->consts[idx];
    rtype_t *vtyp = r_typeof(STACK(robject_t *, src));

    if(!r_subtypep(vtyp, type))
        vm_type_error(vm, "check", "value", type, vtyp);
    DISPATCH();
}
```

26.5 Instruction Macros

Preprocessor macros are defined for binary and unary operations, as well as conversion.

⟨*instruction macros*⟩≡
```
#define BINOP(op,t) PASTE6(arith_,op,_,t,_,t)
#define UNOP(op,t) PASTE4(arith_,op,_,t)
#define CONVOP(a,b) PASTE4(arith_conv_,a,_to_,b)
```

(binary).t *result* $left$ *right*

Computes the binary operation `op` on the values of type `arg_t` specified by the $left$ and *right* operands. The *result* is of type `res_t`.

⟨*instruction macros*⟩+≡
```
#define DO_BINOP(op, arg_t, res_t)                              \
    LABEL(op, arg_t):                                           \
    {                                                           \
        op_stack_t dest = SFETCH(), l = SFETCH(), r = SFETCH(); \
        STACK(R_T(res_t), dest) = BINOP(op,arg_t)               \
            (STACK(R_T(arg_t), l), STACK(R_T(arg_t), r));       \
        DISPATCH();                                             \
    }
```

(unary).t *result arg*

Computes the unary operation `op` on the `arg_t` value at *arg*, producing a `res_t` *result*.

⟨*instruction macros*⟩+≡

```
#define DO_UNOP(op, arg_t, res_t)                              \
    LABEL(op, arg_t):                                          \
    {                                                          \
        op_stack_t dest = SFETCH(), l = SFETCH();              \
        STACK(R_T(res_t), dest) = UNOP(op,arg_t)               \
            (STACK(R_T(arg_t), l));                            \
        DISPATCH();                                            \
    }
```

(convert).a.b *dest src*

Converts the value from *src* of type `a_t` to a value of type `b_t`, and stores it to the stack at *dest*.

⟨*instruction macros*⟩+≡

```
#define DO_CONVOP(a_t, b_t)                                    \
    LABEL(conv, CONV(a_t, b_t)):                               \
    {                                                          \
        op_stack_t dest = SFETCH(), src = SFETCH();            \
        STACK(R_T(b_t), dest) = CONVOP(a_t,b_t)                \
            (STACK(R_T(a_t), src));                            \
        DISPATCH();                                            \
    }
```

26.5.1 Boolean & Conversion

With these, implementations of the conversion and boolean instruction can be expanded.

⟨*instructions*⟩+≡

```
DO_BINOP(and, boolean, boolean)
DO_BINOP(or, boolean, boolean)
DO_UNOP(not, boolean, boolean)
OPS_CONVERSION(DO_CONVOP)
```

26.6 Data Instructions

Instructions with the suffix ".w" have several implementations, one for each distinct width of value upon which they can operate. `op_w` specifies this width, in bits.

⟨*instructions*⟩+≡

```
#define op_w 8
#include "vm/op_data.c.inc"
#undef op_w
#define op_w 32
#include "vm/op_data.c.inc"
#undef op_w
#define op_w 64
#include "vm/op_data.c.inc"
#undef op_w
```

The `op_width_t` macro expands to a C type of the size given by `op_w`.

⟨*op_width_t*⟩≡

```
#define _WIDTH(w) PASTE3(uint, w, _t)
#define op_width_t _WIDTH(op_w)
```

26.6.1 General

mov.w *dest src*

Copies a value from *src* to *dest* in the stack.

⟨*vm/op_data.c.inc*⟩≡
```
LABEL(mov, op_w):
{
    op_stack_t dest = SFETCH(), src = SFETCH();
    STACK(op_width_t, dest) = STACK(op_width_t, src);
    DISPATCH();
}
```

literal.w *dest* ⟨*value*⟩

Copies the literal *value* out of the instruction stream and into the stack at *dest*.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(literal, op_w):
{
    op_stack_t dest = SFETCH();
    op_width_t value = FETCH(op_width_t);
    STACK(op_width_t, dest) = value;
    DISPATCH();
}
```

env.w *dest* [ofs]

Copies the value from byte offset ofs in the current function's closure to the stack at *dest*.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(env, op_w):
{
    op_stack_t dest = SFETCH();
    op_offset_t ofs = OFETCH();
    STACK(op_width_t, dest) = *(op_width_t *)(((uint8_t *)fp->env) + ofs);
    DISPATCH();
}
```

complete.w *dest*

Copies the result value indicated by the preceding `ret` to the stack at *dest*.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(complete, op_w):
{
    op_stack_t dest = SFETCH();
    STACK(op_width_t, dest) = *(op_width_t *)pret;
    DISPATCH();
}
```

getvar.w *dest* [*name*]

Looks up the global variable with the constant *name*, copying its value to *dest*. An error is signalled if it's not defined.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(getvar, op_w):
{
    op_stack_t dest = SFETCH();
    op_offset_t idx = OFETCH();
    rsymbol_t *sym = (rsymbol_t *)fp->fn->consts[idx];
    rglobal_t *global = r_get_global(sym);

    if(!global)
        vm_error(vm, "get", "global `%s` not defined.",
                 r_symstr(sym));
    STACK(op_width_t, dest) = *(op_width_t *)(&global->val);
    DISPATCH();
}
```

setvar.w *src* [*name*] [*def*]

Copies the value from the stack at *src* into the `.val` field of the global variable *name*d by the given constant. If *def* is `true`, this instruction is the initialising definition of a constant, so the variable is flagged `.is_const` and may no longer be altered.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(setvar, op_w):
{
    op_stack_t src = SFETCH();
    op_offset_t idx = OFETCH(), flag = OFETCH();
    rsymbol_t *sym = (rsymbol_t *)fp->fn->consts[idx];
    rglobal_t *global = r_get_global(sym);

    if(!global)
        vm_error(vm, "set", "global `%s` not defined.",
                 r_symstr(sym));
    else if(global->is_const)
        vm_error(vm, "set", "invalid assignment to `const %s`.",
                 r_symstr(sym));
    *(op_width_t *)(&global->val) = STACK(op_width_t, src);
    if(flag)
        global->is_const = true;
    DISPATCH();
}
```

26.6.2 Cells

cellget.w *dest src*

Copies the value out of the `cell` referenced by the pointer at *src* to the stack at *dest*. Essentially, `unbox` without the type-checking.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(cellget, op_w):
{
    op_stack_t dest = SFETCH(), src = SFETCH();
    robject_t *cell = STACK(robject_t *, src);

    STACK(op_width_t, dest) = UNBOX(op_width_t, cell);
    DISPATCH();
}
```

cellset.w *dest src*

Copies the value at *src* into the `cell` referenced by the pointer at *dest*. Cells are mutable, unlike boxes.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(cellset, op_w):
{
    op_stack_t dest = SFETCH(), src = SFETCH();
    robject_t *cell = STACK(robject_t *, dest);

    UNBOX(op_width_t, cell) = STACK(op_width_t, src);
    DISPATCH();
}
```

26.6.3 Element Store

setelt.w *obj src idx*

Stores the value from *src* into the homogeneous vector referenced by the pointer at *obj*. The 1-based index of the destination element is given by the integer at *idx*. If this is outside the vector, or `NA`, the instruction has no effect.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(setelt, op_w):
{
    op_stack_t obj = SFETCH(), src = SFETCH(), idx = SFETCH();
    rbuf_t *buf = STACK(rbuf_t *, obj);
    if(!buf)
        vm_error(vm, "[=", "can't subscript nil.");
    int len = buf->length;
    int i = STACK(rint_t, idx);
    if(!int_na(i) && i > 0 && i <= len)
        ((op_width_t *)buf->elts)[i - 1] = STACK(op_width_t, src);
    DISPATCH();
}
```

setel2.w *obj src row col*

Stores the value at *src* into the homogeneous column-major matrix with reference at *obj*. The 1-based indices of the element to overwrite are specified by the integers on the stack at *row* and *col*. If either are `NA` or outside the bounds of the array, the instruction has no effect. If the array doesn't have `.rank` equal to 2, an error is signalled.

⟨*vm/op_data.c.inc*⟩+≡
```
LABEL(setel2, op_w):
{
    op_stack_t obj = SFETCH(), src = SFETCH(),
               row = SFETCH(), col = SFETCH();
    rarray_t *arr = STACK(rarray_t *, obj);
    int i = STACK(rint_t, row), j = STACK(rint_t, col);
    if(!arr)
        vm_error(vm, "[=", "can't subscript nil.");
    if(arr->rank != 2)
        vm_error(vm, "[=", "array not of rank 2.");
    if(!int_na(i) && i <= arr->shape[0] && i > 0 &&
       !int_na(j) && j <= arr->shape[1] && j > 0)
    {
        unsigned long idx = (unsigned long)arr->shape[0] * (j-1) + (i-1);
        ((op_width_t *)rvec_elts(arr))[idx] = STACK(op_width_t, src);
    }
    DISPATCH();
}
```

26.7 Typed Instructions

The suffix ".t" denotes that the instruction has an implementation for each distinct type of value upon which it operates. `op_t` is set to the base name of the type – scalars `boolean`, `double`, `int`, or object reference `ptr`.

⟨*instructions*⟩+≡
```
#define op_t ptr
#include "vm/op_getelt.c.inc"
#undef op_t
#define op_t double
#include "vm/op_scalar.c.inc"
#include "vm/op_getelt.c.inc"
#include "vm/op_arith.c.inc"
#undef op_t
#define op_t int
#include "vm/op_scalar.c.inc"
#include "vm/op_getelt.c.inc"
#include "vm/op_arith.c.inc"
#undef op_t
#define op_t boolean
#include "vm/op_scalar.c.inc"
#include "vm/op_getelt.c.inc"
#undef op_t
```

The `op_type_t` macro expands to the runtime `typedef` that `op_t` names (Section 19.1).

⟨*op_type_t*⟩≡
```
#define R_T(type) PASTE3(r, type, _t)
#define R_NA(type) PASTE3(r, type, _na)
#define op_type_t R_T(op_t)
```

26.7.1 Arithmetic & Comparison

These are expanded for `double` and `int`.

add.t, **sub**.t, **mul**.t, **div**.t, **pow**.t, **neg**.t

Perform the named arithmetic operation on two scalar values of the same type, resulting in another value of that type.

⟨*vm/op_arith.c.inc*⟩≡
```
DO_BINOP(add, op_t, op_t)
DO_BINOP(sub, op_t, op_t)
DO_BINOP(mul, op_t, op_t)
DO_BINOP(div, op_t, op_t)
DO_BINOP(pow, op_t, op_t)
DO_UNOP(neg, op_t, op_t)
```

lth.t, **lte**.t, **gth**.t, **gte**.t

Compare two scalar values of the same type with an ordered relational operator, yielding a boolean result.

⟨*vm/op_arith.c.inc*⟩+≡
```
DO_BINOP(lth, op_t, boolean)
DO_BINOP(lte, op_t, boolean)
DO_BINOP(gth, op_t, boolean)
DO_BINOP(gte, op_t, boolean)
```

26.7.2 Scalars

These are expanded for all the scalar types.

eql.t, **neq**.t

Compare two scalar values of the same type for equality, inequality, or missingness; with a boolean result.

⟨*vm/op_scalar.c.inc*⟩≡
```
DO_BINOP(eql, op_t, boolean)
DO_BINOP(neq, op_t, boolean)
DO_UNOP(is_na, op_t, boolean)
```

box.t *dest src*

Creates a new box, into which is stored the scalar from *src*; placing a reference to the box on the stack at *dest*.

⟨*vm/op_scalar.c.inc*⟩+≡
```
LABEL(box, op_t):
{
    op_stack_t dest = SFETCH(), src = SFETCH();
    robject_t *obj = r_box_create(RTYPE(op_t));
    UNBOX(op_type_t, obj) = STACK(op_type_t, src);
    STACK(robject_t *, dest) = obj;
    DISPATCH();
}
```

unbox.t *dest src*

Verifies that the object referenced by *src* is a box containing a scalar of the appropriate type. If so (or if it can be converted to one), its contents are extracted and placed at *dest*. If not, an error is signalled.

⟨*vm/op_scalar.c.inc*⟩+≡
```
LABEL(unbox, op_t):
{
    op_stack_t dest = SFETCH(), src = SFETCH();
    robject_t *obj = STACK(robject_t *, src);
    rtype_t *type = r_typeof(obj), *decl = RTYPE(op_t);
    op_type_t *ptr = &STACK(op_type_t, dest);
    if(decl == type)
        *ptr = UNBOX(op_type_t, obj);
    else if(rtype_is_scalar(type))
        scalar_convert(ptr, obj, decl, type);
    else
        vm_type_error(vm, "unbox", "value", decl, type);
    DISPATCH();
}
```

26.7.3 Element Fetch

These are grouped by type, not width, because they require access to an `NA` value.

These are expanded for scalars and also `ptr`, which entails an `op_type_t` of `robject_t *`, and a designated `NA` value of `NULL`.

getelt.t *dest obj idx*

Fetches the element specified by the integer at *idx* from the homogeneous vector referenced by the pointer at *obj*. Its value is stored into the stack at *dest*. An index of `NA`, or outside the vector, yields an `NA` of the appropriate type.

⟨*vm/op_getelt.c.inc*⟩≡
```
LABEL(getelt, op_t):
{
    op_stack_t dest = SFETCH(), obj = SFETCH(), idx = SFETCH();
    rbuf_t *buf = STACK(rbuf_t *, obj);
    if(!buf)
        vm_error(vm, "[", "can't subscript nil.");
    int len = buf->length;
    int i = STACK(rint_t, idx);
    if(!int_na(i) && i <= len && i > 0)
        STACK(op_type_t, dest) = ((op_type_t *)buf->elts)[i - 1];
    else
        STACK(op_type_t, dest) = R_NA(op_t);
    DISPATCH();
}
```

getel2.t *dest obj row col*

Fetches the element specified by the integers at *row* and *col* from the homogeneous column-major matrix referenced by *obj*. Stores, to *dest*, the element's value – or an `NA` if either of the indices are `NA`, or beyond the bounds of the array. If the array *obj* doesn't have `.rank` equal to 2, an error is signalled.

⟨*vm/op_getelt.c.inc*⟩+≡
```
LABEL(getel2, op_t):
{
    op_stack_t dest = SFETCH(), obj = SFETCH(),
               row = SFETCH(), col = SFETCH();
    rarray_t *arr = STACK(rarray_t *, obj);
    int i = STACK(rint_t, row), j = STACK(rint_t, col);
    if(!arr)
        vm_error(vm, "[", "can't subscript nil.");
    if(arr->rank != 2)
        vm_error(vm, "[", "array not of rank 2.");
    if(!int_na(i) && i <= arr->shape[0] && i > 0 &&
       !int_na(j) && j <= arr->shape[1] && j > 0)
    {
        unsigned long idx = (unsigned long)arr->shape[0] * (j-1) + (i-1);
        STACK(op_type_t, dest) = ((op_type_t *)rvec_elts(arr))[idx];
    }
    else
        STACK(op_type_t, dest) = R_NA(op_t);
    DISPATCH();
}
```

26.8 Opcodes

An *opcode* is a member of the `vm_code_t` enumeration with an associated implementation label in `vm_execute`. They are defined with a mess of `cpp` macrology.

⟨*vm/vm_ops.h*⟩≡
```
#define _CAR(a, rest...) a
#define CAR(a) _CAR(a)
⟨opcode macros⟩
⟨opcodes⟩
⟨call subcodes⟩
⟨lambda subcodes⟩
typedef ARRAY(op_code_t) op_code_array_t;
⟨vm_act_rec_t⟩
⟨vm_instr_t⟩
extern const void *const *vm_instr;
```

The `CONV` macro, given type names `a` and `b`, expands to a suitable name for a conversion instruction – "`a2b`". `ENUM` pastes together an opcode name from a base instruction `name` and optional `suffix`. `LABEL` does the same for an implementation label.

⟨*opcode macros*⟩≡
```
#define CONV(a, b) PASTE3(a, 2, b)
#define ENUM(name, suffix...) PASTE3(OP_, name, suffix)
#define LABEL(name, suffix...) PASTE3(do_, name, suffix)
```

The generic instructions have one opcode apiece.

⟨*opcodes*⟩≡
```
#define OPS_GEN                                                    \
        OP_DEF(call_fast), OP_DEF(call_uni), OP_DEF(missing),      \
        OP_DEF(ret), OP_DEF(complete_box), OP_DEF(check),          \
        OP_DEF(jump), OP_DEF(if), OP_DEF(const), OP_DEF(defvar),   \
        OP_DEF(getvar_uni), OP_DEF(setvar_uni), OP_DEF(lambda),    \
        OP_DEF(frame), OP_DEF(geteltptr), OP_DEF(getel2ptr),       \
        OP_DEF(cellmake), OP_DEF(and, boolean), OP_DEF(or, boolean), \
        OP_DEF(not, boolean)
```

Opcodes for the data instructions are grouped by width.

⟨*opcodes*⟩+≡
```
#define OPS_WIDTH(w)                                               \
        OP_DEF(mov, w), OP_DEF(literal, w), OP_DEF(getvar, w),     \
        OP_DEF(setvar, w), OP_DEF(env, w), OP_DEF(complete, w),    \
        OP_DEF(setelt, w), OP_DEF(setel2, w), OP_DEF(cellget, w),  \
        OP_DEF(cellset, w)
```

`op_width_ofs` holds the start opcode for each width group.

⟨*opcodes*⟩+≡
```
#define OPS_WIDTH_ALL OPS_WIDTH(8), OPS_WIDTH(32), OPS_WIDTH(64)
#define OPS_WIDTH_GROUP                          \
    enum { OPS_WIDTH() };                        \
    static const op_code_t op_width_ofs[] = {    \
        CAR(OPS_WIDTH(8)),                       \
        CAR(OPS_WIDTH(32)),                      \
        CAR(OPS_WIDTH(64))                       \
    };
```

Opcodes for the typed instructions are grouped, unsurprisingly, by type.

⟨*opcodes*⟩+≡
```
#define OPS_GETELT(t)                                              \
        OP_DEF(getelt, t), OP_DEF(getel2, t)
#define OPS_SCALAR(t)                                              \
        OP_DEF(box, t), OP_DEF(unbox, t), OP_DEF(eql, t),          \
        OP_DEF(neq, t), OP_DEF(is_na, t)
#define OPS_ARITH(t)                                               \
        OP_DEF(add, t), OP_DEF(sub, t), OP_DEF(mul, t), OP_DEF(div, t), \
        OP_DEF(neg, t), OP_DEF(pow, t), OP_DEF(lth, t), OP_DEF(lte, t), \
        OP_DEF(gth, t), OP_DEF(gte, t)
```

Some scalar types support different sets of instructions.

⟨*opcodes*⟩+≡
```
#define OPS_TYPE_BOOL                            \
    OPS_SCALAR(boolean), OPS_GETELT(boolean)
#define OPS_TYPE_INT                             \
    OPS_SCALAR(int), OPS_GETELT(int), OPS_ARITH(int)
#define OPS_TYPE_DOUBLE                          \
    OPS_SCALAR(double), OPS_GETELT(double), OPS_ARITH(double)
```

The op_type_ofs array holds the start opcode for each type group.

⟨opcodes⟩+≡

```
#define OPS_TYPE_ALL OPS_TYPE_BOOL, OPS_TYPE_INT, OPS_TYPE_DOUBLE
#define OPS_TYPE_GROUP                          \
    enum {                                      \
        OPS_SCALAR(),                           \
        OPS_GETELT(),                           \
        OPS_ARITH()                             \
    };                                          \
    static const op_code_t op_type_ofs[] = {    \
        CAR(OPS_TYPE_BOOL),                     \
        CAR(OPS_TYPE_INT),                      \
        CAR(OPS_TYPE_DOUBLE)                    \
    };
```

op_conv_ofs holds the start opcode of the conversion group.

⟨opcodes⟩+≡

```
#define OPS_CONVERSION(f)                       \
    f(boolean, int)                             \
    f(boolean, double)                          \
    f(int, boolean)                             \
    f(int, double)                              \
    f(double, boolean)                          \
    f(double, int)
#define _CONV(a, b) OP_DEF(conv, CONV(a, b)),
#define OPS_CONV_ALL OPS_CONVERSION(_CONV)
#define OPS_CONV_GROUP                          \
    static const op_code_t op_conv_ofs = CAR(OPS_CONV_ALL)
```

OPS_ALL expands to a list of OP_DEF macros containing every opcode. Defining the latter as ENUM, the vm_opcode_t enumeration can be populated.

⟨opcodes⟩+≡

```
#define OPS_ALL                                 \
    OPS_GEN, OPS_WIDTH_ALL, OPS_TYPE_ALL, OPS_CONV_ALL
#define OP_DEF(def...) ENUM(def)
typedef enum
{
    OPS_ALL
} vm_opcode_t;
OPS_TYPE_GROUP;
OPS_WIDTH_GROUP;
OPS_CONV_GROUP;
#undef OP_DEF
```

Miscellanea

⟨includes⟩≡

```
#include "global.h"
#include "arith/scalar.h"
#include "vm/vm_ops.h"
bool scalar_convert(void *dest, robject_t *obj,
                    rtype_t *to, rtype_t *from); // HACK
```

⟨*vm/vm.h*⟩≡
⟨*vm_ctx_t*⟩
```
void vm_init_ctx(vm_ctx_t *ctx, uint8_t *stack, size_t stacksz);
void vm_fini_ctx(vm_ctx_t *ctx);
void *vm_retain(vm_ctx_t *ctx, void *obj);
void vm_release(vm_ctx_t *ctx, int n);
int vm_execute(vm_ctx_t *vm, rclosure_t *closure);
void vm_error(vm_ctx_t *vm, const char *src, const char *fmt, ...);
```

Part III

Appendices

Appendix A

Runtime Library

This appendix contains miscellaneous builtin functions, including random number generators from R Core Team (2017, Mathlib) and Matsumoto and Nishimura (1998).

rt/math.c

⟨*rt/math.c*⟩≡

```
/*
 *  Mathlib : A C Library of Special Functions
 *  Copyright (C) 1998   Ross Ihaka
 *  Copyright (C) 2000-9 The R Core Team
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, a copy is available at
 *  https://www.R-project.org/Licenses/
 *
*/

#include "global.h"

// mersenne twister

void init_genrand64(unsigned long long seed);
unsigned long long genrand64_int64(void);
double genrand64_real3(void);

void init_rand(unsigned long long seed)
{
    init_genrand64(seed);
}

double unif_rand()
{
```

```
    return genrand64_real3();
}

unsigned unif_rand_u32()
{
    return (unsigned)genrand64_int64();
}

// box-muller rng from MathLib

static double BM_norm_keep = 0.0;

double norm_rand()
{
    if(BM_norm_keep != 0.0)
    { /* An exact test is intentional */
        double s = BM_norm_keep;
        BM_norm_keep = 0.0;
        return s;
    }

    double theta = 2 * M_PI * unif_rand();
    double R = sqrt(-2 * log(unif_rand())) + 10*DBL_MIN; /* ensure non-zero */
    BM_norm_keep = R * sin(theta);
    return R * cos(theta);
}

double rnorm(double mu, double sigma)
{
    if(isnan(mu) || !isfinite(sigma) || sigma < 0)
        return NAN;
    if(sigma == 0 || !isfinite(mu))
        return mu;
    return mu + sigma * norm_rand();
}
```

rt/mt19937-64.c

⟨*rt/mt19937-64.c*⟩≡

```
/*
   A C-program for MT19937-64 (2004/9/29 version).
   Coded by Takuji Nishimura and Makoto Matsumoto.

   This is a 64-bit version of Mersenne Twister pseudorandom number
   generator.

   Before using, initialize the state by using init_genrand64(seed)
   or init_by_array64(init_key, key_length).

   Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura,
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions
   are met:

     1. Redistributions of source code must retain the above copyright
```

```
        notice, this list of conditions and the following disclaimer.

     2. Redistributions in binary form must reproduce the above copyright
        notice, this list of conditions and the following disclaimer in the
        documentation and/or other materials provided with the distribution.

     3. The names of its contributors may not be used to endorse or promote
        products derived from this software without specific prior written
        permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
   A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
   CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
   PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
   LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
   NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
   SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

   References:
   T. Nishimura, ``Tables of 64-bit Mersenne Twisters''
     ACM Transactions on Modeling and
     Computer Simulation 10. (2000) 348--357.
   M. Matsumoto and T. Nishimura,
     ``Mersenne Twister: a 623-dimensionally equidistributed
       uniform pseudorandom number generator''
     ACM Transactions on Modeling and
     Computer Simulation 8. (Jan. 1998) 3--30.

   Any feedback is very welcome.
   http://www.math.hiroshima-u.ac.jp/~m-mat/MT/emt.html
   email: m-mat @ math.sci.hiroshima-u.ac.jp (remove spaces)
*/


#include <stdio.h>

#define NN 312
#define MM 156
#define MATRIX_A 0xB5026F5AA96619E9ULL
#define UM 0xFFFFFFFF80000000ULL /* Most significant 33 bits */
#define LM 0x7FFFFFFFULL /* Least significant 31 bits */


/* The array for the state vector */
static unsigned long long mt[NN];
/* mti==NN+1 means mt[NN] is not initialized */
static int mti=NN+1;

/* initializes mt[NN] with a seed */
void init_genrand64(unsigned long long seed)
{
    mt[0] = seed;
    for (mti=1; mti<NN; mti++)
        mt[mti] =  (6364136223846793005ULL * (mt[mti-1] ^ (mt[mti-1] >> 62)) + mti);
```

```
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
void init_by_array64(unsigned long long init_key[],
                     unsigned long long key_length)
{
    unsigned long long i, j, k;
    init_genrand64(19650218ULL);
    i=1; j=0;
    k = (NN>key_length ? NN : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 62)) * 3935559000370003845ULL))
          + init_key[j] + j; /* non linear */
        i++; j++;
        if (i>=NN) { mt[0] = mt[NN-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=NN-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 62)) * 2862933555777941757ULL))
          - i; /* non linear */
        i++;
        if (i>=NN) { mt[0] = mt[NN-1]; i=1; }
    }

    mt[0] = 1ULL << 63; /* MSB is 1; assuring non-zero initial array */
}

/* generates a random number on [0, 2^64-1]-interval */
unsigned long long genrand64_int64(void)
{
    int i;
    unsigned long long x;
    static unsigned long long mag01[2]={0ULL, MATRIX_A};

    if (mti >= NN) { /* generate NN words at one time */

        /* if init_genrand64() has not been called, */
        /* a default initial seed is used     */
        if (mti == NN+1)
            init_genrand64(5489ULL);

        for (i=0;i<NN-MM;i++) {
            x = (mt[i]&UM)|(mt[i+1]&LM);
            mt[i] = mt[i+MM] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
        }
        for (;i<NN-1;i++) {
            x = (mt[i]&UM)|(mt[i+1]&LM);
            mt[i] = mt[i+(MM-NN)] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
        }
        x = (mt[NN-1]&UM)|(mt[0]&LM);
        mt[NN-1] = mt[MM-1] ^ (x>>1) ^ mag01[(int)(x&1ULL)];

        mti = 0;
    }

    x = mt[mti++];
```

```
    x ^= (x >> 29) & 0x5555555555555555ULL;
    x ^= (x << 17) & 0x71D67FFFEDA60000ULL;
    x ^= (x << 37) & 0xFFF7EEE000000000ULL;
    x ^= (x >> 43);

    return x;
}

/* generates a random number on [0, 2^63-1]-interval */
long long genrand64_int63(void)
{
    return (long long)(genrand64_int64() >> 1);
}

/* generates a random number on [0,1]-real-interval */
double genrand64_real1(void)
{
    return (genrand64_int64() >> 11) * (1.0/9007199254740991.0);
}

/* generates a random number on [0,1)-real-interval */
double genrand64_real2(void)
{
    return (genrand64_int64() >> 11) * (1.0/9007199254740992.0);
}

/* generates a random number on (0,1)-real-interval */
double genrand64_real3(void)
{
    return ((genrand64_int64() >> 12) + 0.5) * (1.0/4503599627370496.0);
}

#if 0
int main(void)
{
    int i;
    unsigned long long init[4]={0x12345ULL, 0x23456ULL,
                                0x34567ULL, 0x45678ULL}, length=4;
    init_by_array64(init, length);
    printf("1000 outputs of genrand64_int64()\n");
    for (i=0; i<1000; i++) {
      printf("%20llu ", genrand64_int64());
      if (i%5==4) printf("\n");
    }
    printf("\n1000 outputs of genrand64_real2()\n");
    for (i=0; i<1000; i++) {
      printf("%10.8f ", genrand64_real2());
      if (i%5==4) printf("\n");
    }
    return 0;
}
#endif
```

rt/math.h

⟨*rt/math.h*⟩≡

```
void init_rand(unsigned long long seed);
double unif_rand();
unsigned unif_rand_u32();
double norm_rand();
double rnorm(double mu, double sigma);
```

rt/opt$_b$*uiltins.h*

⟨*rt/opt_builtins.h*⟩≡

```
const extern builtin_ops_t pure_ops, type_ops, typeof_ops;

typedef struct
{
    int arity;
    union
    {
        rtype_t *(*unfn)(const void *);
        rtype_t *(*binfn)(rtype_t *, rtype_t *);
    };
} type_op_fn_t;

#define type_unop(fn) (&(type_op_fn_t){ .arity = 1, .unfn = fn })
#define type_binop(fn) (&(type_op_fn_t){ .arity = 2, .binfn = fn })
```

rt/builtins.c

⟨*rt/builtins.c*⟩≡

```
#include "global.h"
#include "rt/builtin.h"
#include "rt/opt_builtins.h"
#include <time.h>
#include <sys/resource.h>
#include <sys/time.h>

static void builtin_rm(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rsymbol_t *name;
    } end_args(args, a);
    bool ret = r_unset(a->name);
    //if(!ret)
    //    vm_error(vm, "rm", "can’t remove global ‘%s‘.", r_symstr(a->name));
    //builtin_return(r, robject_t *, NULL);
    builtin_return(r, rboolean_t, ret);
}

static void builtin_print(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        robject_t *x;
    } end_args(args, a);
    argbits_t ab = a->argbits; // TEMP
```

```
    if(!missingp(ab, 0))
        r_print(stdout, a->x);
    fprintf(stdout, "\n");

    builtin_return(r, robject_t *, NULL);
}

static void builtin_eq(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x, *y;
    } end_args(args, a);
    builtin_return(r, rboolean_t, (a->x == a->y));
}

static void builtin_equal(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x, *y;
    } end_args(args, a);
    builtin_return(r, rboolean_t, r_equal(a->x, a->y));
}

// ---

static void builtin_typeof(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *obj;
    } end_args(args, a);
    builtin_return(r, rtype_t *, r_typeof(a->obj));
}

static rtype_t *common_type(rtype_t *self, rtype_t *other)
{
    if(self && other)
        return r_common_type(self, other);
    return NULL;
}

static void builtin_common_type(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rtype_t *self, *other;
    } end_args(args, a);
    builtin_return(r, rtype_t *, common_type(a->self, a->other));
}

static rtype_t *r_ret_type(const void  *arg)
{
    const rtype_t *typ = arg;
    if(rtype_is_callable(typ))
        return typ->sig->ret_type;
    else if(typ == r_type_callable)
        return r_type_object;
    return NULL;
}
```

```
static void builtin_ret_type(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rtype_t *typ;
    } end_args(args, a);
    rtype_t *ret = r_ret_type(a->typ);
    if(!ret)
        vm_error(vm, "ret_type", "invalid argument.");
    builtin_return(r, rtype_t *, ret);
}

static rtype_t *r_elt_type(const void *arg)
{
    const rtype_t *typ = arg;
    if(rtype_is_container(typ))
        return typ->elt;
    else if(typ == r_type_vector || typ == r_type_array)
        return r_type_object;
    return NULL;
}

static void builtin_elt_type(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rtype_t *typ;
    } end_args(args, a);
    rtype_t *ret = r_elt_type(a->typ);
    if(!ret)
        vm_error(vm, "elt_type", "invalid argument.");
    builtin_return(r, rtype_t *, ret);
}

// ---

static void builtin_subtype(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rtype_t *self, *other;
    } end_args(args, a);
    if(!a->self || !a->other)
        vm_error(vm, "subtype", "invalid argument.");
    builtin_return(r, rboolean_t, r_subtypep(a->self, a->other));
}

static void builtin_compat(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rtype_t *self, *other;
    } end_args(args, a);
    if(!a->self || !a->other)
        vm_error(vm, "compat", "invalid argument.");
    builtin_return(r, rint_t, r_type_compat(a->self, a->other, true));
}

static void builtin_hash(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
```

```
        robject_t *x;
    } end_args(args, a);
    builtin_return(r, rint_t, r_hash(a->x));
}

static inline double tv_to_sec(struct timeval *tv)
{
    return tv->tv_sec + tv->tv_usec/(double)1000000;
}

// { real, user, sys } times in seconds
static void builtin_time(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    rvector_t *res = vm_retain(vm, rvec_create(r_type_vec_double, 3));
    rdouble_t *times = rvec_elts(res);
    rsymbol_t **names = rvec_elts(rvec_add_names(res));
    struct rusage ru;
    struct timeval tv;

    getrusage(RUSAGE_SELF, &ru);
    gettimeofday(&tv, NULL);
    times[0] = tv_to_sec(&tv);
    times[1] = tv_to_sec(&ru.ru_utime);
    times[2] = tv_to_sec(&ru.ru_stime);
    names[0] = r_intern("real");
    names[1] = r_intern("user");
    names[2] = r_intern("system");
    builtin_return(r, rvector_t *, res);
    vm_release(vm, 1);
}

#define clamp_int(v) ((v) > INT_MAX ? rint_na : (rint_t)(v))

static void builtin_gc(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    rvector_t *res = vm_retain(vm, rvec_create(r_type_vec_int, 4));
    rint_t *counts = rvec_elts(res);
    rsymbol_t **names = rvec_elts(rvec_add_names(res));
    gc_stats_t stats;

    gc_collect();
    gc_collect_stats(&stats);
    counts[0] = clamp_int(stats.ncollect);
    counts[1] = clamp_int(stats.nchunks);
    counts[2] = clamp_int(stats.obj_bytes/1024.0);
    counts[3] = clamp_int(stats.vec_bytes/1024.0);
    names[0] = r_intern("collections");
    names[1] = r_intern("chunks");
    names[2] = r_intern("object KB");
    names[3] = r_intern("vector KB");
    builtin_return(r, rvector_t *, res);
    vm_release(vm, 1);
}

static void builtin_rand(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rint_t min, max;
```

```
    } end_args(args, a);

    if(a->max == rint_na || a->min == rint_na || a->max <= a->min)
        vm_error(vm, "rand", "invalid argument.");
    unsigned range = (unsigned)((int64_t)a->max - a->min) + 1;
    builtin_return(r, rint_t, (rint_t)(unif_rand_u32() % range + a->min));
}

static void builtin_srand(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        rint_t seed; // optional
    } end_args(args, a);
    argbits_t ab = a->argbits; // TEMP
    unsigned long long seed;

    if(!missingp(ab, 0))
    {
        if(a->seed == rint_na)
            vm_error(vm, "srand", "invalid argument.");
        seed = (unsigned)a->seed;
    }
    else
    {
        time_t tval = time(NULL);
        if(tval == (time_t)-1)
            vm_error(vm, "srand", "time(2) error."); // wat.
        seed = tval;
    }
    init_rand(seed);
    builtin_return(r, robject_t *, NULL);
}

static void builtin_source(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rstring_t *name;
    } end_args(args, a);
    rclosure_t *cl;

    // the compiler is somewhat cavalier about managed pointers
    gc_set_enabled(false);
    // read and compile file
    cl = source(a->name->string);
    // closure is rooted on return
    gc_set_enabled(true);
    builtin_return(r, rclosure_t *, cl);
}

static void builtin_stop(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rstring_t *msg;
    } end_args(args, a);
    vm_error(vm, "stop", "%s", a->msg ? a->msg->string : "");
    // does not return
}
```

```
static void builtin_assert(vm_ctx_t *vm, rbuiltin_t *fn,
                           uint8_t *args, void *r)
{
    def_args {
        rboolean_t test;
    } end_args(args, a);
    if(a->test != true)
        vm_error(vm, "assert", "assertion failed.");
    builtin_return(r, robject_t *, NULL);
}

const builtin_init_t runtime_builtins[] = {
    defbuiltin("rm", builtin_rm, &r_type_boolean,
               { "name", &r_type_symbol }),
    defcbuiltin("eq", builtin_eq, &r_type_boolean,
               &pure_ops, NULL, // pointers are immutable
               { "x", &r_type_object },
               { "y", &r_type_object }),
    defbuiltin("equal", builtin_equal, &r_type_boolean,
               { "x", &r_type_object },
               { "y", &r_type_object }),
    defbuiltin("print", builtin_print, &r_type_object,
               { "x", &r_type_object, true }),
    defbuiltin("rt_time", builtin_time, &r_type_object,
               { }),
    defbuiltin("rand", builtin_rand, &r_type_int,
               { "min", &r_type_int },
               { "max", &r_type_int }),
    defbuiltin("srand", builtin_srand, &r_type_object,
               { "seed", &r_type_int, true }),
    defcbuiltin("typeof", builtin_typeof, &r_type_type,
               &typeof_ops, NULL,
               { "obj", &r_type_object }),
    defcbuiltin("common_type", builtin_common_type, &r_type_type,
               &type_ops, type_binop(common_type),
               { "self", &r_type_type },
               { "other", &r_type_type }),
    defcbuiltin("ret_type", builtin_ret_type, &r_type_type,
               &type_ops, type_unop(r_ret_type),
               { "type", &r_type_type }),
    defcbuiltin("elt_type", builtin_elt_type, &r_type_type,
               &type_ops, type_unop(r_elt_type),
               { "type", &r_type_type }),
    defcbuiltin("subtype", builtin_subtype, &r_type_boolean,
               &pure_ops, NULL, // types are immutable
               { "self", &r_type_type },
               { "other", &r_type_type }),
    defcbuiltin("compat", builtin_compat, &r_type_int,
               &pure_ops, NULL, // types are immutable
               { "self", &r_type_type },
               { "other", &r_type_type }),
    defbuiltin("hash", builtin_hash, &r_type_int,
               { "x", &r_type_object }),
    defbuiltin("gc_collect", builtin_gc, &r_type_object,
               { }),
    defbuiltin("source", builtin_source, &r_type_callable,
               { "file", &r_type_string }),
```

```
    defbuiltin("stop", builtin_stop, &r_type_object,
               { "message", &r_type_string }),
    defbuiltin("assert", builtin_assert, &r_type_object,
               { "expr", &r_type_boolean }),
    { }
};
```

rt/opt_builtins.c

⟨rt/opt_builtins.c⟩≡

```
#include "global.h"
#include "rt/builtin.h"
#include "rt/opt_builtins.h"
#include "ir.h"
#include "opt.h"
#include "vm/vm_ops.h"
#include "gen_code.h"

static cresult trans_typeof(opt_phase phase, cnode_t *node, cell_t *cell,
                            const cbuiltin_t *bi, cnode_array_t *args)
{
    assert(node->type == CN_CALL || node->type == CN_CALL_FAST);
    if(alen(args) != 1)
        return SUCCESS;
    // the condition for typeof could be weakened slightly, to something like
    // "if the static type is the most precise possible dynamic type",
    // but that isn't expressible here.
    if(phase == TRANSFER && opt.opt_constfold)
    {
        cell_t *ac = cell_for(aref(args,0));
        if(!cell_const_obj(ac) && !cell_const_lambda(ac))
                return SUCCESS;

        cell_set_const(cell, (robject_t *)cell_type(ac));
        // DBG("%s folds to ", bi->name);
        // r_print(stderr, cell_type(ac));
        // DBG("\n");
    }
    return SUCCESS;
}

static cresult trans_type(opt_phase phase, cnode_t *node, cell_t *cell,
                          const cbuiltin_t *bi, cnode_array_t *args)
{
    type_op_fn_t *tofn = bi->data;
    assert(node->type == CN_CALL || node->type == CN_CALL_FAST);
    assert(tofn);

    if(alen(args) != tofn->arity)
        return SUCCESS;
    if(phase == TRANSFER && opt.opt_constfold)
    {
        cnode_t *arg;
        array_foreach_entry(args, arg)
        {
            cell_t *ac = cell_for(arg);
            if(!cell_const_obj(ac) || cell_type(ac) != r_type_type)
```

```
                return SUCCESS;
        }

        rtype_t *ret = (tofn->arity == 1)
            ? tofn->unfn(cell_const(cell_for(aref(args,0))))
            : tofn->binfn(cell_const(cell_for(aref(args,0))),
                          cell_const(cell_for(aref(args,1))));
        if(ret)
        {
            cell_set_const(cell, (robject_t *)ret);
            // DBG("%s folds to ", bi->name);
            // r_print(stderr, ret);
            // DBG("\n");
        }
    }
    return SUCCESS;
}

// XXX need a user-facing ‘missing‘ special form: arg parsed as
// identifier; symbol must name an arg bound by the current function
// convert emits a CN_BUILTIN directly -- like the prologue does in
// ir_convert
static inline cresult fold_missing(cell_t *cell, cell_t *bits, argbits_t idx)
{
    if(opt.opt_constfold && cell_const_obj(bits))
    {
        bool val = argbits_missing(UNBOX(argbits_t, cell_const(bits)), idx);
        cell_set_const(cell, c_intern(r_box(r_type_boolean, &val)));
        return CHANGED;
    }
    return SUCCESS;
}

static cresult trans_missing(opt_phase phase, cnode_t *node, cell_t *cell,
                             const cbuiltin_t *bi, cnode_array_t *args)
{
    assert(node->type == CN_BUILTIN);
    assert(node->builtin.optype && r_typeof(node->builtin.optype) == r_type_int);
    assert(alen(args) == 1);

    cell_t *bits = cell_for(aref(args, 0));
    robject_t *idx = (robject_t *)node->builtin.optype; // this is a hack

    assert(cell_type(bits) == r_type_int);
    if(phase == TRANSFER)
    {
        cell_set_type(cell, r_type_boolean);
        return fold_missing(cell, bits, UNBOX(argbits_t, idx));
    }
    // transform; already a builtin, so don’t care
    return SUCCESS;
}

static void gen_missing(cnode_t *node)
{
    // MISSING dest, bits, idx
    asm_op(OP_missing);
    asm_stack(loc_for(node));
```

```
    asm_stack(loc_for(aref(&node->builtin.args, 0)));
    // we stashed a box in .optype. retrieve it and emit the value as a literal operand
    assert(node->builtin.optype && r_typeof(node->builtin.optype) == r_type_int);
    asm_offset(UNBOX(argbits_t, node->builtin.optype));
}

static bool always(cnode_t *node, bool may_alias)
    { return true; }

const cbuiltin_t missing = {
    &(builtin_ops_t) { trans_missing, NULL, gen_missing,
                       .is_pure = always },
    NULL, "missing"
};

const builtin_ops_t pure_ops = { .is_pure = always };
const builtin_ops_t typeof_ops = { trans_typeof, NULL, NULL, .is_pure = always };
const builtin_ops_t type_ops = { trans_type, NULL, NULL, .is_pure = always };
```

rt/options.h

⟨*rt/options.h*⟩≡

```
typedef struct __attribute__ ((__packed__))
{
    // optimisation
    bool opt_builtin;
    bool opt_inline;
    bool opt_dce;
    bool opt_constfold;
    bool opt_dvn;
    // debugging
    bool dbg_dump_ir;
    // printing
    int print_digits;
    int print_line;
    int print_max;
    int print_name_max;
} options_t;
extern options_t opt;
void rt_init_options();
```

rt/options.c

⟨*rt/options.c*⟩≡

```
#include "global.h"
#include <ctype.h>
#include "rt/builtin.h"
#include "arith/scalar.h"
#include "arith/vec.h"

options_t opt;

static inline void *opt_ofs(argdesc_t *arg)
{
    return (uint8_t *)&opt + arg->offset - sizeof(argbits_t);
}
```

```
// return a named vector containing the compiler options
static rvector_t *get_options(vm_ctx_t *vm, funsig_t *sig)
{
    int nargs = sig->nargs, i;
    rvector_t *res = vm_retain(vm, rvec_create(r_type_vec_object, nargs));
    robject_t **elts = rvec_elts(res);

    for(i=0; i<nargs; i++)
    {
        argdesc_t *arg = &sig->args[i];
        void *src = opt_ofs(arg);

        if(rtype_is_scalar(arg->type))
            elts[i] = vm_retain(vm, r_box(arg->type, src));
        else
            elts[i] = *(robject_t **)src;
    }
    // ensure the vector is fully initialised before calling rvec_add_names...
    rsymbol_t **names = rvec_elts(rvec_add_names(res));
    for(i=0; i<nargs; i++)
        names[i] = sig->args[i].name;
    vm_release(vm, 1+nargs);
    return res;
}

// set the given compiler options, returning the previous state
// so it can be reset with something like ‘apply(options, state)‘
static void builtin_options(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    funsig_t *sig = rcall_sig(&fn->base);
    robject_t *res = vm_retain(vm, get_options(vm, sig));
    args -= sig->argsz; // XXX builtin call convention is silly, fix this imo
    argbits_t argbits = *(argbits_t *)args;
    assert(sizeof(opt) == sig->argsz - sizeof(argbits_t));

    for(int i=0; i<sig->nargs; i++)
    {
        if(argbits & (1<<i))
        {
            argdesc_t *arg = &sig->args[i];
            void *src = args + arg->offset;
            void *dest = opt_ofs(arg);

            if(rtype_is_scalar(arg->type)
               && scalar_is_na(arg->type, src) == true)
            {   // XXX vm_warning?
                fprintf(stderr, "warning: argument ‘%s‘ not valid.\n",
                        r_symstr(arg->name));
                continue;
            }
            memcpy(dest, src, rtype_eltsz(arg->type));
        }
    }
    vm_release(vm, 1);
    builtin_return(r, robject_t *, res);
}
```

```
// XXX multiexpand to avoid these getting out of sync
// (typedef in rt/options.h also)
const builtin_init_t options_builtin =
    defbuiltin("options", builtin_options, &r_type_vec_object,
               { "opt_builtin", &r_type_boolean, true },
               { "opt_inline", &r_type_boolean, true },
               { "opt_dce", &r_type_boolean, true },
               { "opt_constfold", &r_type_boolean, true },
               { "opt_dvn", &r_type_boolean, true },
               { "dbg_dump_ir", &r_type_boolean, true },
               { "print_digits", &r_type_int, true },
               { "print_line", &r_type_int, true },
               { "print_max", &r_type_int, true },
               { "print_name_max", &r_type_int, true });

/*
static char *upcase(char *str)
{
    static char buf[32];
    int sz = sizeof(buf)-1;
    char *ptr = buf;

    while(*str != '\0' && sz-- > 0)
        *ptr++ = toupper(*str++);
    *ptr = '\0'; // terminate
    return buf;
}
*/

static int init_option(char *name, int def)
{
    char *val = getenv(name);
    if(val)
        return atoi(val);
    return def;
}

void rt_init_options()
{
    opt = (options_t) {
        .opt_builtin = init_option("OPT_BUILTIN", true),
        .opt_inline = init_option("OPT_INLINE", true),
        .opt_dce = init_option("OPT_DCE", true),
        .opt_constfold = init_option("OPT_CONSTFOLD", true),
        .opt_dvn = init_option("OPT_DVN", true),
        .dbg_dump_ir = init_option("DBG_DUMP_IR", false),
        .print_digits = 6,
        .print_line = 76,
        .print_max = 200,
        .print_name_max = 15,
    };

    rbuiltin_define(&options_builtin);
/*
    rbuiltin_t *fn = (rbuiltin_t *)rbuiltin_define(&options_builtin);
    funsig_t *sig = fn->base.base.type->sig;
    for(int i=0; i<sig->nargs; i++)
    {
```

```
        argdesc_t *arg = &sig->args[i];
        char *str = getenv(upcase(r_symstr(arg->name)));
        if(str)
        {
            void *dest = opt_ofs(arg);
            // ... atoi ...
        }
    }
*/
}
```

Appendix B

Arithmetic Library

This appendix contains an arithmetic and support library for scalars, vectors and arrays; together with the builtin functions that make them available to users of the system. They contain minimal significant innovation, and an unpleasant amount of M4 (GNU M4).

arith/arith.m4

⟨*arith/arith.m4*⟩≡

```
m4_changecom(‘M4’, ‘’)
m4_changequote(‘{{’, ‘}}’)

M4 bind the first argument (name) as an element of the second (quoted list)
M4 and expand the third argument (quotation) for each one in order

m4_define({{shift3}}, {{m4_shift(m4_shift(m4_shift($1)))}})
m4_define({{_foreach}}, {{
  m4_ifelse({{$#}},{{3}},,{{
    m4_define({{$1}}, {{$3}})
    $2
    $0({{$1}},{{$2}},shift3({{$@}}))
  }})
}})
m4_define({{foreach}}, {{m4_pushdef({{$1}})_foreach({{$1}},{{$3}},$2,)m4_popdef({{$1}})}})

M4 usual list routines
m4_define({{car}}, {{$1}})
m4_define({{cdr}}, {{m4_shift($@)}})
m4_define({{cadr}}, {{car(cdr($@))}})

M4 token-pasting
m4_define({{paste}}, {{m4_ifelse({{$#}},1,{{$1}},{{$1_{{}}$0(m4_shift($@))}})}})

M4 diversion wrapper
m4_define({{stash}},{{m4_divert($1){{}}$2{{}}m4_divert}})

M4 names of things
m4_define({{make_type}}, {{r$1_t}})
m4_define({{make_type_obj}}, {{r_type_$1}})
m4_define({{check_na}}, {{$2_na($1)}})
m4_define({{make_na}}, {{r$1_na}})
m4_define({{make_min}}, {{r$1_min}})
m4_define({{make_max}}, {{r$1_max}})
```

```
M4 static arithmetic promotion: expand to the argument found first in ‘types‘
M4 FIXME should be ifelse(car($3), ‘$1’, match one, car($3), ‘$2’, match two, no match)
m4_define({{_make_rtype}}, {{m4_ifelse({{$1}}, car($3), {{$1}},
{{m4_ifelse({{$2}}, car($3), {{$2}},
  {{$0({{$1}}, {{$2}}, {{cdr($3)}})}})}})}})
m4_define({{make_rtype}}, {{_make_rtype($1, $2, {{scalar_types}})}})

M4 convenient synonyms
m4_define({{xtype_t}}, {{make_type(xtype)}})
m4_define({{ytype_t}}, {{make_type(ytype)}})
m4_define({{rtype}}, {{make_rtype(xtype, ytype)}})
m4_define({{rtype_t}}, {{make_type(rtype)}})
m4_define({{op_name}}, {{car(op_spec)}})
m4_define({{op_sym}}, {{cadr(op_spec)}})

M4 m4_define({{op_binary}}, {{$1 cadr(op_spec) $2}})
m4_define({{op_binary}}, {{m4_ifelse(op_sym,{{}},
  {{op_name{{}}($1, $2)}},
  {{$1 op_sym $2}})}})

M4 m4_define({{op_unary}}, {{cadr(op_spec) $1}})
m4_define({{op_unary}}, {{m4_ifelse(op_sym,{{}},
  {{op_name{{}}($1)}},
  {{op_sym $1}})}})

m4_define({{op_conv}}, {{$2_conv($1)}})
M4 double ops have no overflow
m4_define({{op_xflow}}, {{m4_ifelse(rtype,{{double}},0,
  {{op_name{{}}_xflow($1, $2)}})}})
m4_define({{conv_func}}, {{paste({{arith}}, {{conv}}, xtype, {{to}}, ytype)}})
m4_define({{binary_func}}, {{paste({{arith}}, op_name, xtype, ytype)}})
m4_define({{unary_func}}, {{paste({{arith}}, op_name, xtype)}})

M4 loops
m4_define({{for_ops}}, {{foreach(op_spec, {{$1}}, {{$2}})}})
m4_define({{for_type}}, {{foreach(xtype, {{$1}}, {{$2}})}})
m4_define({{for_types}}, {{
  foreach(xtype, {{$1}}, {{
  foreach(ytype, {{$1}}, {{
  $2
  }})
  }})
}})

M4 configuration
m4_define({{element_types}}, {{double, int, boolean, ptr}})
m4_define({{scalar_types}}, {{double, int, boolean}})
m4_define({{numeric_types}}, {{double, int}})

M4 on numeric types, return promoted
m4_define({{arith_ops}}, {{{{add, +}}, {{sub, -}},
  {{mul, *}}, {{div, /}}, {{pow}}}})
M4 on numeric types, return same
m4_define({{arith_unary_ops}}, {{{{neg, -}}}})
M4 on numeric types, return same, no analogous symbol
M4 use cadr(op_spec) for identity instead
m4_define({{arith_reduce_ops}}, {{{{add, 0}}, {{mul, 1}}}})
```

```
M4 on numeric types, return boolean
m4_define({{rela_ops}}, {{{{lth, <}}, {{lte, <=}}, {{gth, >}}, {{gte, >=}}}})

M4 on scalar types, return boolean
m4_define({{eql_ops}}, {{{{eql, ==}}, {{neq, !=}}}})
M4 on scalar types, return boolean
M4 none for now
M4 m4_define({{eql_unary_ops}}, {{{{something}}}})

M4 on booleans only
m4_define({{bool_ops}}, {{{{and, &}}, {{or, |}}}})
M4 on booleans only
m4_define({{bool_unary_ops}}, {{{{not, !}}}})
M4 on booleans only, etc, etc
m4_define({{bool_reduce_ops}}, {{{{and, true}}, {{or, false}}}})

M4 on doubles only
m4_define({{math_ops}}, {{{{atan2}}}})
m4_define({{math_unary_ops}}, {{{{sqrt}}, {{sin}}, {{cos}}, {{tan}},
                                {{asin}}, {{acos}}, {{atan}},
                                {{log}}, {{exp}}, {{round}}}})
```

arith/scalar.h.in

⟨*arith/scalar.h.in*⟩≡

```
/*
  m4_include(‘arith/arith.m4’)
*/

// return true if the value is NA
#define boolean_na(v) ((v) == rboolean_na)
#define int_na(v) ((v) == rint_na)
#define double_na(v) (isnan(v))
// convert the value to the type
#define boolean_conv(v) ((v) != 0)
#define double_conv(v) (rdouble_t)(v)
#define int_conv(v) ((((v) > INT_MAX) | ((v) <= INT_MIN)) ? rint_na : (rint_t)(v))

// do not take ‘r‘: compute it explicitly and hope that common
// subexpression elimination does its thing.
// return true if the operation over- or underflowed
#define add_xflow(x, y) ((x > 0) ^ (y < (x + y)))
#define signs(a, b) ((a < 0) ^ (b < 0))
#define sub_xflow(x, y) (signs(x, y) & signs(x, (x - y)))
#define mul_xflow(x, y) (((int64_t)x * (int64_t)y) != (x * y))
#define div_xflow(x, y) (y==0)
// XXX inefficient
#define pow_xflow(x, y) (fabs(pow(x,y)) > (INT_MAX-1))

// m4_define({{expand_binary}}, {{
static inline rtype_t paste(arith, op_name, xtype, ytype)(xtype_t x, ytype_t y)
{
    return (check_na(x, xtype) | check_na(y, ytype) | op_xflow(x, y))
        ? make_na(rtype)
        : op_binary(x, y);
}
```

```
// }})
// m4_define({{expand_unary}}, {{
static inline rtype_t paste(arith, op_name, xtype)(xtype_t x)
{
    rtype_t r = op_unary(x);
    return check_na(x, xtype) ? make_na(rtype) : r;
}
// }})
```

Arithmetic

⟨*arith/scalar.h.in*⟩+≡

```
// M4 prefer native double NA handling, thanks to C arithmetic conversion
// m4_pushdef({{double_na}}, {{0}})

// for_types({{numeric_types}}, {{
// for_ops({{arith_ops}}, {{
expand_binary()
// }})
// }})

// for_type({{numeric_types}}, {{
// for_ops({{arith_unary_ops}}, {{
expand_unary()
// }})
// }})

// M4 for other ops, we must explicitly check
// m4_popdef({{double_na}})
// M4 overflow doesn't occur outside arithmetic
// m4_pushdef({{op_xflow}}, {{0}})
```

Relational

⟨*arith/scalar.h.in*⟩+≡

```
// M4 ops from now on return boolean
// m4_pushdef({{rtype}}, {{boolean}})

// for_types({{numeric_types}}, {{
// for_ops({{rela_ops}}, {{
expand_binary()
// }})
// }})
```

Equality

⟨*arith/scalar.h.in*⟩+≡

```
// for_types({{scalar_types}}, {{
// for_ops({{eql_ops}}, {{
expand_binary()
// }})
// }})
```

Booleans

⟨*arith/scalar.h.in*⟩+≡
```
// m4_pushdef({{xtype}}, {{boolean}}) m4_pushdef({{ytype}}, {{boolean}})
// for_ops({{bool_ops}}, {{
expand_binary()
// }})
// for_ops({{bool_unary_ops}}, {{
expand_unary()
// }})
// m4_popdef({{xtype}}) m4_popdef({{ytype}})
```

NA

⟨*arith/scalar.h.in*⟩+≡
```
// for_type({{scalar_types}}, {{
static inline rboolean_t paste(arith, is_na, xtype)(xtype_t x)
{
    return check_na(x, xtype);
}
// }})

// M4 reset defs to default
// m4_popdef({{rtype}})
// m4_popdef({{op_xflow}})
```

Conversion

⟨*arith/scalar.h.in*⟩+≡
```
// for_types({{scalar_types}}, {{
// m4_ifelse(xtype, ytype, {{}}, {{
static inline ytype_t paste(arith, conv, xtype, to, ytype)(xtype_t x)
{
    return (check_na(x, xtype)) ? make_na(ytype) : op_conv(x, ytype);
}
// }})
// }})
```

Box & Unbox

⟨*arith/scalar.h.in*⟩+≡
```
// for_type({{scalar_types}}, {{
static inline xtype_t paste(arith, conv, ptr, to, xtype)(robject_t *x)
{
    if(r_typeof(x) != make_type_obj(xtype))
        return make_na(xtype);
    return UNBOX(xtype_t, x);
}

static inline robject_t *paste(arith, conv, xtype, to, ptr)(xtype_t x)
{
    robject_t *box = r_box_create(make_type_obj(xtype));
    UNBOX(xtype_t, box) = x;
    return box;
}
// }})
```

Dispatch Index Codes

⟨*arith/scalar.h.in*⟩+≡
```
enum {
    CODE_boolean = SC_BOOLEAN,
    CODE_int = SC_INT,
    CODE_double = SC_DOUBLE,
    CODE_MAX_SCALAR,
    CODE_ptr = CODE_MAX_SCALAR,
    CODE_MAX
};
enum {
    CODE_sca, CODE_vec
};
```

arith/vec.h.in

⟨*arith/vec.h.in*⟩≡
```
/*
  m4_include(`arith/arith.m4')
*/
// {{
typedef void (*binop_fn)(void *, void *, void *, int);
typedef void (*unop_fn)(void *, void *, int);
typedef void (*conv_fn)(void *, void *, int);
typedef void (*fill_na_fn)(void *, size_t );
typedef void (*idx_fn)(void *, void *, void *, int, int, int);
// }}

// m4_define({{expand_bintab}}, {{
extern const binop_fn paste(op_name, funcs)[2][2][CODE_MAX_SCALAR][CODE_MAX_SCALAR];
// }})
// m4_define({{expand_untab}}, {{
extern const unop_fn paste(op_name, funcs)[2][CODE_MAX_SCALAR];
// }})
// m4_define({{expand_redtab}}, {{
extern const unop_fn paste(reduce, op_name, funcs)[CODE_MAX_SCALAR];
// }})
// m4_define({{expand_convtab}}, {{
extern const conv_fn paste(conv, funcs)[2][CODE_MAX_SCALAR][CODE_MAX_SCALAR];
// }})
```

Externs

⟨*arith/vec.h.in*⟩+≡
```
// for_ops({{arith_ops, rela_ops, eql_ops, bool_ops, math_ops}}, {{
expand_bintab()
// }})
// for_ops({{arith_unary_ops, bool_unary_ops, math_unary_ops, {{is_na}}}}, {{
expand_untab()
// }})
// for_ops({{arith_reduce_ops, bool_reduce_ops}}, {{
expand_redtab()
// }})
expand_convtab()
```

Typedefs

FIXME should be a `binop_subtab [CODE_MAX_SCALAR][CODE_MAX_SCALAR]`, because if one is `NULL`, they all are.

⟨*arith/vec.h.in*⟩+≡
```
// {{
typedef const binop_fn binop_fntab[2][2][CODE_MAX_SCALAR][CODE_MAX_SCALAR];
typedef const unop_fn unop_fntab[2][CODE_MAX_SCALAR];
typedef const unop_fn reduce_fntab[CODE_MAX_SCALAR];
```

Inlines

⟨*arith/vec.h.in*⟩+≡
```
// return the element type for an operand, given an aggregate or scalar type.
// returns NULL if typ isn't a scalar or an aggregate containing scalars.
static inline rtype_t *arith_elt_type(rtype_t *typ)
{
    if(rtype_is_container(typ))
        typ = typ->elt;
    return rtype_is_scalar(typ) ? typ : NULL;
}

// recycle src along dest
// slen/dlen lengths in elements, sz element size in bytes
static inline
void copy_recycle(void *dest, void *src, int dlen, int slen, size_t sz)
{
    if(slen == 0)
    {
        memset(dest, 0, sz * dlen);
        return;
    }
    if(slen >= dlen)
    {
        memcpy(dest, src, sz * dlen);
        return;
    }

    ptrdiff_t adv = slen * sz;
    uint8_t *end = (uint8_t *)dest + dlen * sz;
    for(uint8_t *ptr = dest; ptr < end; ptr += adv)
        memcpy(ptr, src, min(adv, end - ptr));
}

static inline rboolean_t scalar_is_na(rtype_t *type, void *src)
{
    assert(rtype_is_scalar(type));
    rboolean_t res;
    unop_fn is_na = is_na_funcs[CODE_sca][rscal_code(type)];
    is_na(&res, src, 1);
    return res;
}
```

Prototypes

⟨*arith/vec.h.in*⟩+≡
```
robject_t *arith_binary(vm_ctx_t *vm, binop_fntab fntab, bool pred,
                        robject_t *xobj, robject_t *yobj);
robject_t *arith_unary(vm_ctx_t *vm, unop_fntab fntab, bool pred,
                       robject_t *xobj);
robject_t *arith_convert(vm_ctx_t *vm, robject_t *obj, rtype_t *to);
robject_t *arith_reduce(vm_ctx_t *vm, reduce_fntab fntab, robject_t *xobj);
bool scalar_convert(void *dest, robject_t *obj, rtype_t *to, rtype_t *from);

robject_t *vec_fetch(vm_ctx_t *vm, robject_t *obj, robject_t *idx);
robject_t *vec_store(vm_ctx_t *vm, robject_t *obj, robject_t *val,
                     robject_t *idx);
robject_t *arr_fetch(vm_ctx_t *vm, robject_t *obj, robject_t *idx,
                     robject_t **idxs, int nidxs);
robject_t *arr_store(vm_ctx_t *vm, robject_t *obj, robject_t *val,
                     robject_t *idx, robject_t **idxs, int nidxs);
void copy_names(robject_t *dest, robject_t *src);
void rt_install_vectors();
// }}
```

arith/dispatch.h

⟨*arith/dispatch.h*⟩≡
```
// builtin data
typedef struct
{
    const void *fntab; // table of function pointers
    bool is_pred; // whether this builtin a predicate
    op_code_t op; // vm instruction opcode
} arith_builtin_t;

// binary operation dispatch
typedef struct
{
    rtype_t *typ, *optyp;
    size_t xsz, ysz, rsz;
    scalar_code xcode, ycode;
    bool xvec, yvec; // 'is array or vector', really
    bool xarr, yarr; // 'is array and not vector'
} binop_chk_t;

bool check_binary(binop_chk_t *chk, rtype_t *xt, rtype_t *yt, bool pred);

static inline binop_fn dispatch_binary(binop_chk_t *chk, binop_fntab fntab)
{
    return fntab[chk->xvec][chk->yvec][chk->xcode][chk->ycode];
}

// unary operation dispatch
typedef struct
{
    rtype_t *typ;
    scalar_code code;
    bool vec, arr; // same as for binop
} unop_chk_t;
```

```
bool check_unary(unop_chk_t *chk, rtype_t *xt, bool pred);

static inline unop_fn dispatch_unary(unop_chk_t *chk, unop_fntab fntab)
{
    return fntab[chk->vec][chk->code];
}

// reduction dispatch
typedef struct
{
    rtype_t *typ; // always a scalar
    scalar_code code;
    bool vec;
} reduce_chk_t;

bool check_reduce(reduce_chk_t *chk, rtype_t *xt);

static inline unop_fn dispatch_reduce(reduce_chk_t *chk, reduce_fntab fntab)
{
    return fntab[chk->code];
}

// conversion dispatch
typedef struct
{
    rtype_t *rt;
    bool vec, arr, generic;
    scalar_code fromcode, tocode;
} conv_chk_t;

// from could be anything; to is always a scalar
bool check_conv(conv_chk_t *chk, rtype_t *from, rtype_t *to);

static inline conv_fn dispatch_conv(conv_chk_t *chk)
{
    return conv_funcs[chk->vec][chk->fromcode][chk->tocode];
}

// collection fetch and store

// scalar, vector, boolean, or omitted index
typedef enum { SINGLE, INDEX, MASK, MISSING } idx_kind;

// vector
typedef struct
{
    idx_kind kind;
    bool takeptr; // use replacement/result value directly, do not indirect
    bool names; // fetch element names
    rtype_t *typ; // result type/replacement element type
} idx_chk_t;

bool check_fetch(idx_chk_t *chk, rtype_t *typ, rtype_t *ityp);
bool check_store(idx_chk_t *chk, rtype_t *typ, rtype_t *ityp, rtype_t *rtyp);

// array
typedef struct
```

```
{
    idx_kind *kinds; // one per index, _including_ the first
    bool takeptr;
    int rank; // rank of indexed value - 0: scalar, 1: vector, >1: array
    rtype_t *typ;  // element type of result/replacement
} arr_chk_t;

bool check_arr_fetch(arr_chk_t *achk, rtype_t *typ, rtype_t **ityps,
                     int ntyps);
bool check_arr_store(arr_chk_t *achk, rtype_t *typ, rtype_t **ityps,
                     int ntyps, rtype_t *rtyp);
```

arith/vec.c.in

⟨*arith/vec.c.in*⟩≡

```
/*
  m4_include(‘arith/arith.m4’)
*/
#include "global.h"
#include "arith/scalar.h"
#include "arith/vec.h"
#include "arith/dispatch.h"
/*
  m4_define({{vec_binary_func}}, {{paste({{vec}}, op_name, xtype, $1, ytype, $2)}})
  m4_define({{vec_unary_func}}, {{paste({{vec}}, op_name, xtype, $1)}})
  m4_define({{vec_conv_func}}, {{paste({{vec_conv}}, $1, xtype, to, ytype)}})
  m4_define({{code}},{{paste(CODE,$1)}})
*/
// m4_define({{expand_bintab}}, {{
const binop_fn paste(op_name, funcs)[2][2][CODE_MAX_SCALAR][CODE_MAX_SCALAR] = {
    m4_undivert(1)
};
// }})
// m4_define({{expand_untab}}, {{
const unop_fn paste(op_name, funcs)[2][CODE_MAX_SCALAR] = {
    m4_undivert(1)
};
// }})
// m4_define({{expand_redtab}}, {{
const unop_fn paste(reduce, op_name, funcs)[CODE_MAX_SCALAR] = {
    m4_undivert(1)
};
// }})
// m4_define({{expand_convtab}}, {{
const conv_fn paste(conv, funcs)[2][CODE_MAX_SCALAR][CODE_MAX_SCALAR] = {
    m4_undivert(1)
};
// }})

// m4_define({{expand_vec}}, {{
static void vec_binary_func(v, v)(void *r, void *x, void *y, int len)
{
    xtype_t *vx = x;
    ytype_t *vy = y;
    rtype_t *vr = r;
    for(int i=0; i < len; i++)
        vr[i] = binary_func()(vx[i], vy[i]);
```

```
}

static void vec_binary_func(v, s)(void *r, void *x, void *y, int len)
{
    xtype_t *vx = x;
    ytype_t vy = *(ytype_t *)y;
    rtype_t *vr = r;
    for(int i=0; i < len; i++)
        vr[i] = binary_func()(vx[i], vy);
}

static void vec_binary_func(s, v)(void *r, void *x, void *y, int len)
{
    xtype_t vx = *(xtype_t *)x;
    ytype_t *vy = y;
    rtype_t *vr = r;
    for(int i=0; i < len; i++)
        vr[i] = binary_func()(vx, vy[i]);
}

static void vec_binary_func(s, s)(void *r, void *x, void *y, int len)
{
    xtype_t *vx = x;
    ytype_t *vy = y;
    rtype_t *vr = r;
    *vr = binary_func()(*vx, *vy);
}

/*
  stash(1,{{
  [CODE_sca][CODE_sca][code(xtype)][code(ytype)] = vec_binary_func(s,s),
  [CODE_sca][CODE_vec][code(xtype)][code(ytype)] = vec_binary_func(s,v),
  [CODE_vec][CODE_sca][code(xtype)][code(ytype)] = vec_binary_func(v,s),
  [CODE_vec][CODE_vec][code(xtype)][code(ytype)] = vec_binary_func(v,v),
  }})
*/
// }})

// m4_define({{expand_vec_unary}}, {{
static void vec_unary_func(v)(void *r, void *x, int len)
{
    xtype_t *vx = x;
    rtype_t *vr = r;
    for(int i=0; i < len; i++)
        vr[i] = unary_func()(vx[i]);
}

static void vec_unary_func(s)(void *r, void *x, int len)
{
    xtype_t *vx = x;
    rtype_t *vr = r;
    *vr = unary_func()(*vx);
}

/*
  stash(1,{{
  [CODE_sca][code(xtype)] = vec_unary_func(s),
  [CODE_vec][code(xtype)] = vec_unary_func(v),
```

```
  }})
*/
// }})

// m4_define({{expand_vec_reduce}}, {{
static void paste(vec, reduce, op_name, xtype)(void *r, void *x, int len)
{
    xtype_t vr = op_sym;
    xtype_t *vx = x;
    for(int i=0; i < len; i++)
        vr = paste(arith, op_name, xtype, xtype)(vr, vx[i]);
    *(xtype_t *)r = vr;
}

/*
  stash(1,{{
  [code(xtype)] = paste(vec, reduce, op_name, xtype),
  }})
*/
// }})
```

Arithmetic

⟨*arith/vec.c.in*⟩+≡

```
// for_ops({{arith_ops}}, {{
// for_types({{numeric_types}}, {{
expand_vec()
// }})
expand_bintab()
// }})

// for_ops({{arith_unary_ops}}, {{
// for_type({{numeric_types}}, {{
expand_vec_unary()
// }})
expand_untab()
// }})

// for_ops({{arith_reduce_ops}}, {{
// for_type({{numeric_types}}, {{
expand_vec_reduce()
// }})
expand_redtab()
// }})
```

Relational

⟨*arith/vec.c.in*⟩+≡

```
// m4_pushdef({{rtype}}, {{boolean}})

// for_ops({{rela_ops}}, {{
// for_types({{numeric_types}}, {{
expand_vec()
// }})
expand_bintab()
// }})
```

Equality

⟨*arith/vec.c.in*⟩+≡
```
// for_ops({{eql_ops}}, {{
// for_types({{scalar_types}}, {{
expand_vec()
// }})
expand_bintab()
// }})
```

NA

⟨*arith/vec.c.in*⟩+≡
```
// for_ops({{{{is_na}}}}, {{
// for_type({{scalar_types}}, {{
expand_vec_unary()
// }})
expand_untab()
// }})
```

Booleans

⟨*arith/vec.c.in*⟩+≡
```
// m4_pushdef({{xtype}}, {{boolean}})
// m4_pushdef({{ytype}}, {{boolean}})
// for_ops({{bool_ops}}, {{
expand_vec()
expand_bintab()
// }})
// for_ops({{bool_unary_ops}}, {{
expand_vec_unary()
expand_untab()
// }})
// for_ops({{bool_reduce_ops}}, {{
expand_vec_reduce()
expand_redtab()
// }})
// m4_popdef({{xtype}}) m4_popdef({{ytype}}) m4_popdef({{rtype}})
```

Special Functions

⟨*arith/vec.c.in*⟩+≡
```
// m4_pushdef({{xtype}}, {{double}})
// m4_pushdef({{ytype}}, {{double}})
// m4_pushdef({{rtype}}, {{double}})
// m4_pushdef({{binary_func}}, {{op_name{}}}})
// m4_pushdef({{unary_func}}, {{op_name{}}}})
// for_ops({{math_ops}}, {{
expand_vec()
expand_bintab()
// }})
// for_ops({{math_unary_ops}}, {{
expand_vec_unary()
expand_untab()
// }})
// m4_popdef({{binary_func}}) m4_popdef({{unary_func}})
// m4_popdef({{xtype}}) m4_popdef({{ytype}}) m4_popdef({{rtype}})
```

Conversion

⟨*arith/vec.c.in*⟩+≡

```
// for_types({{scalar_types}}, {{
// m4_ifelse(xtype, ytype, {{}}, {{
static void vec_conv_func(v)(void *dest, void *src, int len)
{
    xtype_t *vs = src;
    ytype_t *vd = dest;
    for(int i=0; i < len; i++)
        vd[i] = conv_func()(vs[i]);
}
static void vec_conv_func(s)(void *dest, void *src, int len)
{
    *(ytype_t *)dest = conv_func()(*(xtype_t *)src);
}
/*
  stash(1,{{
  [CODE_sca][code(xtype)][code(ytype)] = vec_conv_func(s),
  [CODE_vec][code(xtype)][code(ytype)] = vec_conv_func(v),
  }})
*/
// }})
// }})
expand_convtab()
```

Store

⟨*arith/vec.c.in*⟩+≡

```
/*
  m4_define({{maybe_conv}},
    {{m4_ifelse(xtype,ytype,$1,
      {{paste(arith,conv,ytype,to,xtype)}}{{($1)}})}})
*/
#define NEXT1(p, start, end) ((p+1 >= end) ? start : p+1)
// for_types({{element_types}}, {{

// v[ints] = r
// recycle r along idx
// idxs which are out of range are ignored
static void
paste(int, set, xtype, from, ytype)(void *r, void *v, void *idx,
                                    int rlen, int vlen, int ilen)
{
    rint_t *vi = idx;
    xtype_t *vv = v;
    ytype_t *vr = r;
    ytype_t *rstart = vr, *rend = vr + rlen;

    for(int i=0; i<ilen; i++)
    {
        rint_t j = vi[i];
        if(!(int_na(j) || j > vlen || j <= 0))
        {
            vv[j-1] = maybe_conv(*vr);
            vr = NEXT1(vr, rstart, rend);
        }
    }
```

```
}

// v[bools] = r
// recycles r along idx, and idx along v
static void
paste(bool, set, xtype, from, ytype)(void *r, void *v, void *idx,
                                     int rlen, int vlen, int ilen)
{
    rboolean_t *vi = idx;
    rboolean_t *istart = vi, *iend = vi + ilen;
    xtype_t *vv = v;
    ytype_t *vr = r;
    ytype_t *rstart = vr, *rend = vr + rlen;

    for(int i=0; i<vlen; i++)
    {
        rboolean_t b = *vi;
        vi = NEXT1(vi, istart, iend);
        if(b == true)
        {
            vv[i] = maybe_conv(*vr);
            vr = NEXT1(vr, rstart, rend);
        }
    }
}

// v[] = r
static void
paste(missing, set, xtype, from, ytype)(void *r, void *v, void *idx,
                                        int rlen, int vlen, int ilen)
{
    xtype_t *vv = v;
    ytype_t *vr = r;
    ytype_t *rstart = vr, *rend = vr + rlen;

    for(int i=0; i<vlen; i++)
    {
        vv[i] = maybe_conv(*vr);
        vr = NEXT1(vr, rstart, rend);
    }
}

/*
  stash(1,{{
  [code(xtype)][code(ytype)] = paste(int, set, xtype, from, ytype),
  }})
  stash(2,{{
  [code(xtype)][code(ytype)] = paste(bool, set, xtype, from, ytype),
  }})
  stash(3,{{
  [code(xtype)][code(ytype)] = paste(missing, set, xtype, from, ytype),
  }})
*/
// }})

const idx_fn int_set_funcs[CODE_MAX][CODE_MAX] = {
    //m4_undivert(1)
};
```

```
const idx_fn bool_set_funcs[CODE_MAX][CODE_MAX] = {
    //m4_undivert(2)
};
const idx_fn missing_set_funcs[CODE_MAX][CODE_MAX] = {
    //m4_undivert(3)
};
```

Fetch

⟨*arith/vec.c.in*⟩+≡

```
// for_type({{element_types}}, {{
static void paste(fill_na, xtype)(void *r, size_t sz)
{
    xtype_t *vr = r;
    int len = sz / sizeof(*vr);
    for(int i=0; i < len; i++)
        vr[i] = make_na(xtype);
}

// ... = v[ints]
// r is sized correctly
static void paste(int, get, xtype)(void *r, void *v, void *idx,
                                   int rlen, int vlen, int ilen)
{
    rint_t *vi = idx;
    xtype_t *vv = v;
    xtype_t *vr = r;

    // for each element in the index
    for(int i=0; i < ilen; i++)
    {
        rint_t j = vi[i];
        // if the index element is out of range or NA, the result is NA,
        // else it's the indexed element from the vector
        if(int_na(j) || j > vlen || j <= 0)
            vr[i] = make_na(xtype);
        else
            vr[i] = vv[j-1];
    }
}

// ... = v[bools]
// recycle idx along v
// r is sized correctly, but may need padding with NAs if v is shorter than idx
static void paste(bool, get, xtype)(void *r, void *v, void *idx,
                                    int rlen, int vlen, int ilen)
{
    rboolean_t *vi = idx;
    rboolean_t *istart = vi, *iend = vi + ilen;
    xtype_t *vv = v;
    xtype_t *vr = r;

    for(int i=0; i<vlen; i++)
    {
        rboolean_t b = *vi;
        vi = NEXT1(vi, istart, iend);
        // if it's not false
        if(b != false)
```

```
        {
            // set the current result element - to NA, if the index elt is;
            // or to the corresponding vector element
            *vr++ = boolean_na(b) ? make_na(xtype) : vv[i];
        }
    }

    // if the index is longer than the vector, fill the rest of the result with NAs
    if(vlen < ilen)
    {
        uint8_t *end = (uint8_t *)((xtype_t *)r + rlen);
        paste(fill_na, xtype)(vr, end - (uint8_t *)vr);
    }
}

// ... = v[]
static void paste(missing, get, xtype)(void *r, void *v, void *idx,
                                       int rlen, int vlen, int ilen)
{
    xtype_t *vv = v;
    xtype_t *vr = r;

    for(int i=0; i<vlen; i++)
        vr[i] = vv[i];
}

/*
  stash(1,{{
  [code(xtype)] = paste(int, get, xtype),
  }})
  stash(2,{{
  [code(xtype)] = paste(bool, get, xtype),
  }})
  stash(3,{{
  [code(xtype)] = paste(missing, get, xtype),
  }})
  stash(4,{{
  [code(xtype)] = paste(fill_na, xtype),
  }})
*/
// }})

const idx_fn int_get_funcs[CODE_MAX] = {
    //m4_undivert(1)
};
const idx_fn bool_get_funcs[CODE_MAX] = {
    //m4_undivert(2)
};
const idx_fn missing_get_funcs[CODE_MAX] = {
    //m4_undivert(3)
};
const fill_na_fn fill_na_funcs[CODE_MAX] = {
    //m4_undivert(4)
};
```

Operations

⟨*arith/vec.c.in*⟩+≡

```
//{{

// helpers
// FIXME use rvec_len/rvec_elts now they exist
static inline int arith_length(robject_t *obj)
{
    if(!obj)
        return 0;
    if(!rtype_is_container(r_typeof(obj)))
        return 1;
    return ((rbuf_t *)obj)->length;
}

static inline void *arith_storage(robject_t *obj)
{
    if(!obj)
        return NULL;
    if(rtype_is_scalar(r_typeof(obj)))
        return BOXPTR(obj);
    assert(rtype_is_container(r_typeof(obj)));
    return ((rbuf_t *)obj)->elts;
}

// FIXME some generalised new() kind of thing, plx
static inline robject_t *arith_create(rtype_t *typ, int len)
{
    if(rtype_is_scalar(typ))
        return r_box_create(typ);
    else if(rtype_is_vector(typ))
        return (robject_t *)rvec_create(typ, len);
    else if(rtype_is_array(typ))
        return (robject_t *)rarr_create_buf(typ, len);
    assert(!"can't arith_create that type");
    return NULL;
}

// dest and src are the same (container) type
// may allocate; retain unrooted inputs.
// do not invoke before initialising freshly allocated vectors of reference type
// despite the name, will share .names/.dimnames pointer if at all possible
void copy_names(robject_t *dest, robject_t *src)
{
    rtype_t *dtype = r_typeof(dest);
    if(rtype_is_vector(dtype))
    {
        rvector_t *dvec = (rvector_t *)dest, *svec = (rvector_t *)src;

        if(rvec_len(dvec) == rvec_len(svec))
        {
            // copy the reference if they're the same length
            dvec->names = svec->names;
        }
        else
        {
            rvec_add_names(dvec);
            copy_recycle(rvec_elts(dvec->names), rvec_elts(svec->names),
```

```
                        rvec_len(dvec), rvec_len(svec),
                        sizeof(rsymbol_t *));
        }
    }
    else if(rtype_is_array(dtype))
    {
        rarray_t *darr = (rarray_t *)dest, *sarr = (rarray_t *)src;
        assert(rarr_conform(darr, sarr));
        // arrays always conform, so just copy the reference
        darr->dimnames = sarr->dimnames;
    }
}

static inline void copy_shape(robject_t *robj, robject_t *obj)
{
    rarray_t *arr = (rarray_t *)obj;
    rarr_set_shape((rarray_t *)robj, arr->rank, arr->shape);
}

bool check_binary(binop_chk_t *chk, rtype_t *xt, rtype_t *yt, bool pred)
{
    *chk = (binop_chk_t) {
        .xvec = rtype_is_container(xt), .yvec = rtype_is_container(yt),
        .xarr = rtype_is_array(xt), .yarr = rtype_is_array(yt)
    };

    // take the element type if arg is a container
    rtype_t *xet = chk->xvec ? xt->elt : xt,
            *yet = chk->yvec ? yt->elt : yt;

    // only work on scalars
    if(!rtype_is_scalar(xet) || !rtype_is_scalar(yet))
        return false;

    // dispatch at the higher of the arguments
    chk->optyp = rscal_promote(xet, yet);

    // result element type is dispatch type, or boolean if op is a predicate
    rtype_t *etyp = pred ? r_type_boolean : chk->optyp;

    // result is an array if either arg is, else a vector if either arg is, else a scalar
    if(chk->xarr || chk->yarr)
        chk->typ = rarr_type_create(etyp);
    else if(chk->xvec || chk->yvec)
        chk->typ = rvec_type_create(etyp);
    else
        chk->typ = etyp;

    // scalar attributes cached for later use
    chk->xsz = rscal_size(xet);
    chk->ysz = rscal_size(yet);
    chk->rsz = rscal_size(etyp);
    chk->xcode = rscal_code(xet);
    chk->ycode = rscal_code(yet);
    return true;
}

// XXX ancillae, surely?
```

```
static void copy_ancilla_binary(binop_chk_t *chk, robject_t *robj, robject_t *xobj,
                                robject_t *yobj, bool y_long)
{
    bool xn = is_named(xobj), yn = is_named(yobj);

    // if an array is involved
    if(chk->xarr || chk->yarr)
    {
        // take its shape
        if(chk->xarr)
            copy_shape(robj, xobj);
        else if(chk->yarr)
            copy_shape(robj, yobj);
        // take dimnames if present
        if(chk->xarr && xn)
            copy_names(robj, xobj);
        else if(chk->yarr && yn)
            copy_names(robj, yobj);
    }
    // neither x nor y are arrays
    else if(chk->xvec && xn)
    {
        // x is a named vector
        // if y is a named vector longer than x, use its names
        if(chk->yvec && yn && y_long)
            copy_names(robj, yobj);
        else // else use x's
            copy_names(robj, xobj);
    }
    // x is not a named vector, and y is
    else if(chk->yvec && yn)
        copy_names(robj, yobj);
}

robject_t *arith_binary(vm_ctx_t *vm, binop_fntab fntab, bool pred,
                        robject_t *xobj, robject_t *yobj)
{
    rtype_t *xt = r_typeof(xobj), *yt = r_typeof(yobj);
    binop_chk_t chk = { 0 };

    if(!check_binary(&chk, xt, yt, pred))
        vm_error(vm, "arith_binary", "incompatible argument types.");

    rtype_t *rt = vm_retain(vm, chk.typ);
    binop_fn fn = dispatch_binary(&chk, fntab);

    if(!fn)
        vm_error(vm, "arith_binary", "operation not supported.");
    if(chk.xarr & chk.yarr && !rarr_conform((rarray_t *)xobj, (rarray_t *)yobj))
        vm_error(vm, "arith_binary", "non-conforming arrays.");

    int xlen = arith_length(xobj), ylen = arith_length(yobj);
    int rlen = (xlen == 0 || ylen == 0) ? 0 : max(xlen, ylen);
    robject_t *robj = vm_retain(vm, arith_create(rt, rlen));
    void *x = arith_storage(xobj), *y = arith_storage(yobj),
         *r = arith_storage(robj);

    // copy shape and dim/names
```

```
    copy_ancilla_binary(&chk, robj, xobj, yobj, ylen > xlen);
    // if either are length 0, the result is length 0, an R-ism
    if(chk.xvec && chk.yvec && rlen > 0)
    {
        bool xlong = xlen >= ylen;
        int roundlen = xlong ? ylen : xlen;
        int rounds = rlen / roundlen;
        int extra = rlen - rounds * roundlen;

        ptrdiff_t xadv = xlong ? (chk.xsz * ylen) : 0;
        ptrdiff_t yadv = xlong ? 0 : (chk.ysz * xlen);
        ptrdiff_t radv = xlong ? (chk.rsz * ylen) : (chk.rsz * xlen);

        for(int n=0; n < rounds; n++)
        {
            fn(r, x, y, roundlen);
            r = (uint8_t *)r + radv;
            x = (uint8_t *)x + xadv;
            y = (uint8_t *)y + yadv;
        }
        if(extra > 0)
        {
            fn(r, x, y, extra);
            // vm_warning(vm, "non-integral recycling");
        }
    }
    else // since rlen is correct, and arith_storage does the right thing
        fn(r, x, y, rlen);
    vm_release(vm, 2);
    return robj;
}

bool check_unary(unop_chk_t *chk, rtype_t *xt, bool pred)
{
    *chk = (unop_chk_t) {
        .vec = rtype_is_container(xt),
        .arr = rtype_is_array(xt),
    };
    rtype_t *xet = chk->vec ? xt->elt : xt;

    if(!rtype_is_scalar(xet))
        return false;

    rtype_t *etyp = pred ? r_type_boolean : xet;

    if(chk->arr)
        chk->typ = rarr_type_create(etyp);
    else if(chk->vec)
        chk->typ = rvec_type_create(etyp);
    else
        chk->typ = etyp;
    chk->code = rscal_code(xet);
    return true;
}

static void copy_ancilla_unary(bool arr, robject_t *robj, robject_t *xobj)
{
    if(arr)
```

```
        copy_shape(robj, xobj);
    if(is_named(xobj))
        copy_names(robj, xobj);
}

robject_t *arith_unary(vm_ctx_t *vm, unop_fntab fntab, bool pred,
                       robject_t *xobj)
{
    rtype_t *xt = r_typeof(xobj);
    unop_chk_t chk = { 0 };

    if(!check_unary(&chk, xt, pred))
        vm_error(vm, "arith_unary", "incompatible argument type.");

    rtype_t *rt = vm_retain(vm, chk.typ);
    unop_fn fn = dispatch_unary(&chk, fntab);

    if(!fn)
        vm_error(vm, "arith_unary", "operation not supported.");

    int len = arith_length(xobj);
    robject_t *robj = vm_retain(vm, arith_create(rt, len));
    void *x = arith_storage(xobj), *r = arith_storage(robj);

    copy_ancilla_unary(chk.arr, robj, xobj);
    fn(r, x, len);
    vm_release(vm, 2);
    return robj;
}


bool check_reduce(reduce_chk_t *chk, rtype_t *xt)
{
    *chk = (reduce_chk_t) {
        .vec = rtype_is_container(xt),
    };
    rtype_t *xet = chk->vec ? xt->elt : xt;

    if(!rtype_is_scalar(xet))
        return false;
    chk->typ = xet;
    chk->code = rscal_code(xet);
    return true;
}

robject_t *arith_reduce(vm_ctx_t *vm, reduce_fntab fntab, robject_t *xobj)
{
    rtype_t *xt = r_typeof(xobj);
    reduce_chk_t chk = { 0 };

    if(!check_reduce(&chk, xt))
        vm_error(vm, "arith_reduce", "incompatible argument type.");
    if(!chk.vec)
        return xobj;

    int len = arith_length(xobj);
    rtype_t *rt = vm_retain(vm, chk.typ);
    unop_fn fn = dispatch_reduce(&chk, fntab);
```

```
    if(!fn)
        vm_error(vm, "arith_reduce", "operation not supported.");

    robject_t *robj = vm_retain(vm, r_box_create(rt));
    void *x = arith_storage(xobj), *r = arith_storage(robj);

    fn(r, x, len);
    vm_release(vm, 2);
    return robj;
}

// convert a vector of objects to a single scalar type
static inline void vec_conv_generic(rtype_t *to, void *dest, void *src, int len)
{
    assert(rtype_is_scalar(to));
    size_t sz = rtype_eltsz(to);
    uint8_t *ptr = dest;
    robject_t **vs = src;
    fill_na_fn na_fn = fill_na_funcs[rscal_code(to)];

    for(int i=0; i < len; i++, ptr+=sz)
    {
        robject_t *elt = vs[i];
        rtype_t *from = r_typeof(elt);

        // if the element is of the correct type already, unbox it
        if(from == to)
            r_unbox(ptr, elt);
        // if it's another scalar, convert it
        else if(rtype_is_scalar(from))
        {
            conv_fn fn = conv_funcs[0][rscal_code(from)][rscal_code(to)];
            fn(ptr, BOXPTR(elt), 1);
        }
        // it's not a value, so NA
        else
            na_fn(ptr, sz);
    }
}

bool check_conv(conv_chk_t *chk, rtype_t *from, rtype_t *to)
{
    *chk = (conv_chk_t) {
        .vec = rtype_is_container(from),
        .arr = rtype_is_array(from),
    };
    assert(rtype_is_scalar(to));
    if(chk->vec)
    {
        chk->rt = chk->arr
            ? rarr_type_create(to)
            : rvec_type_create(to);

        if(rtype_is_scalar(from->elt))
            chk->fromcode = rscal_code(from->elt);
        else
            chk->generic = true;
```

```
    }
    else
    {
        chk->rt = to;
        if(!rtype_is_scalar(from))
            return false;
        chk->fromcode = rscal_code(from);
    }
    chk->tocode = rscal_code(to);
    return true;
}

// convert a vector or scalar to have a given scalar (element) type
robject_t *arith_convert(vm_ctx_t *vm, robject_t *obj, rtype_t *to)
{
    rtype_t *typ = r_typeof(obj);
    rtype_t *from = arith_elt_type(typ);
    conv_chk_t chk = { 0 };

    if(!check_conv(&chk, typ, to))
        vm_error(vm, "arith_convert", "invalid argument type.");
    // XXX we might want to copy instead, because:
    // y = as_double(x); y[...] = crap; shouldn't sometimes fill x with crap?
    if(to == from)
        return obj;

    rtype_t *rt = vm_retain(vm, chk.rt);

    if(!rt)
        vm_error(vm, "arith_convert", "invalid argument type.");

    int len = arith_length(obj);
    robject_t *robj = vm_retain(vm, arith_create(rt, len));
    void *x = arith_storage(obj), *r = arith_storage(robj);

    copy_ancilla_unary(chk.arr, robj, obj);

    if(chk.generic)
    {
        vec_conv_generic(to, r, x, len);
    }
    else
    {
        conv_fn fn = dispatch_conv(&chk);

        if(!fn)
            vm_error(vm, "arith_convert", "invalid argument type.");
        fn(r, x, len);
    }
    vm_release(vm, 2);
    return robj;
}

// now used in both compiler and vm
bool scalar_convert(void *dest, robject_t *obj, rtype_t *to, rtype_t *from)
{
    assert(rtype_is_scalar(from));
    assert(rtype_is_scalar(to));
```

```
    assert(to != from);

    conv_fn fn = conv_funcs[0][rscal_code(from)][rscal_code(to)];

    if(!fn)
        return false;
    fn(dest, arith_storage(obj), 1);
    return true;
}


static inline int idx_func_code(rtype_t *typ)
{
    return rtype_is_scalar(typ) ? rscal_code(typ) : CODE_ptr;
}

static inline idx_fn idx_access_func(idx_kind kind, int rcode, int vcode)
{
    switch(kind)
    {
    case SINGLE:
    case INDEX:
        return (rcode == -1) ?
            int_get_funcs[vcode] : int_set_funcs[vcode][rcode];
    case MASK:
        return (rcode == -1) ?
            bool_get_funcs[vcode] : bool_set_funcs[vcode][rcode];
    case MISSING:
        return (rcode == -1) ?
            missing_get_funcs[vcode] : missing_set_funcs[vcode][rcode];
    }
    return NULL; /* NOTREACHED */
}

static inline void
idx_copy_names(rsymbol_t **rnames, rsymbol_t **vnames, void *i, idx_kind kind,
               int rlen, int vlen, int ilen)
{
    idx_fn fn = idx_access_func(kind, -1, CODE_ptr);
    fn(rnames, vnames, i, rlen, vlen, ilen);
}

static int count_nonfalse(rboolean_t *elts, int n)
{
    int c = 0;
    for(int i=0; i<n; i++)
        c += (elts[i] != false);
    return c;
}

// given a mask of length ilen, return the length of the vector that
// will result when it's recycled over a vector of length vlen.
// (note that this may be > vlen, per semantics)
static int mask_result_len(int vlen, rboolean_t *mask, int ilen)
{
    if(ilen == 0)
        return 0;
    if(ilen >= vlen)
```

```
        return count_nonfalse(mask, ilen);

    int sum = 0;
    for(int i=0; i<vlen; i += ilen)
        sum += count_nonfalse(mask, min(ilen, vlen - i));
    return sum;
}

static inline bool check_index(idx_kind *kind, rtype_t *ityp)
{
    if(ityp == r_type_nil)
        *kind = MISSING;
    else if(arith_elt_type(ityp) == r_type_int)
        *kind = rtype_is_scalar(ityp) ? SINGLE : INDEX;
    else if(arith_elt_type(ityp) == r_type_boolean)
        *kind = MASK;
    else
        return false; // no other valid index types
    return true;
}

bool check_fetch(idx_chk_t *chk, rtype_t *typ, rtype_t *ityp)
{
    if(!rtype_is_vector(typ) && !rtype_is_array(typ))
        return false;
    if(!check_index(&chk->kind, ityp))
        return false;
    if(chk->kind == SINGLE)
    {
        // return a single element
        chk->typ = typ->elt;
        // single element of reference type can be returned directly
        if(!rtype_is_scalar(typ->elt))
            chk->takeptr = true;
    }
    else if(rtype_is_array(typ))
    {
        // indexing into an array with a single vectorised subscript
        // always returns a vector result (this differs from R)
        chk->typ = rvec_type_create(typ->elt);
        // (and never returns element names, since arrays don't have them)
    }
    else
    {
        chk->typ = typ;
        chk->names = true;
    }
    return true;
}

robject_t *vec_fetch(vm_ctx_t *vm, robject_t *val, robject_t *idx)
{
    rtype_t *typ = r_typeof(val), *ityp = r_typeof(idx);
    idx_chk_t chk = { 0 };

    if(!check_fetch(&chk, typ, ityp))
        vm_error(vm, "[", "invalid argument type.");
```

```
    rtype_t *rt = vm_retain(vm, chk.typ);
    int vlen = arith_length(val), ilen = arith_length(idx);
    bool names = chk.names & is_named(val);
    int rlen = 0;

    if(chk.kind == MASK)
        rlen = mask_result_len(vlen, arith_storage(idx), ilen);
    else if(chk.kind == MISSING)
        rlen = vlen;
    else
        rlen = ilen;

    robject_t *res = NULL;
    void *r;

    if(chk.takeptr)
        r = &res;
    else
    {
        res = arith_create(rt, rlen);
        r = arith_storage(res);
    }
    vm_retain(vm, res);
    if(rlen > 0)
    {
        idx_fn fn = idx_access_func(chk.kind, -1, idx_func_code(typ->elt));
        void *v = arith_storage(val), *i = arith_storage(idx);

        fn(r, v, i, rlen, vlen, ilen);
        if(names)
        {
            rvector_t *vvec = (rvector_t *)val, *rvec = (rvector_t *)res;

            rvec_add_names(rvec);
            idx_copy_names(rvec_elts(rvec->names), rvec_elts(vvec->names), i,
                           chk.kind, rlen, vlen, ilen);
        }
    }
    vm_release(vm, 2);
    return res;
}

bool check_store(idx_chk_t *chk, rtype_t *typ, rtype_t *ityp, rtype_t *rtyp)
{
    rtype_t *ret, *et;

    if(!rtype_is_vector(typ) && !rtype_is_array(typ))
        return false;
    if(!check_index(&chk->kind, ityp))
        return false;
    // replacement check
    if(rtype_is_vector(rtyp) && chk->kind != SINGLE) // vector replacement?
    {
        // replace with contents: check against element type
        ret = rtyp->elt;
    }
    else // either replacement not a vector, or single value required
    {
```

```
        // replace with value
        ret = rtyp;
        // replace with the reference as a value
        if(!rtype_is_scalar(rtyp))
            chk->takeptr = true;
    }
    // container is always a vector
    et = typ->elt;
    // storing into
    if(rtype_is_scalar(et))
    {
        // a vector of scalars
        // FIXME set_xxx_from_ptr is a thing now, maybe use that
        if(!rtype_is_scalar(ret)) // can't accept reference objects
            return false;
    }
    else
    {
        // a vector of objects
        if(!r_subtypep(ret, et)) // can't accept incompatible objects
            return false;
    }
    chk->typ = ret;
    return true;
}

robject_t *vec_store(vm_ctx_t *vm, robject_t *val, robject_t *rpl,
                     robject_t *idx)
{
    rtype_t *typ = r_typeof(val), *ityp = r_typeof(idx), *rtyp = r_typeof(rpl);
    idx_chk_t chk = { 0 };
    void *r;

    if(!check_store(&chk, typ, ityp, rtyp))
        vm_error(vm, "[=", "invalid argument type.");

    // vector[...]
    int vlen = arith_length(val), ilen = arith_length(idx), rlen;

    if(chk.takeptr)
    {
        r = &rpl;
        rlen = 1;
    }
    else
    {
        r = arith_storage(rpl);
        rlen = arith_length(rpl);
    }
    if(rlen > 0 && (ilen > 0 || chk.kind == MISSING))
    {
        idx_fn fn = idx_access_func(chk.kind, idx_func_code(chk.typ),
                                    idx_func_code(typ->elt));
        void *v = arith_storage(val), *i = arith_storage(idx);

        fn(r, v, i, rlen, vlen, ilen);
    }
    return rpl;
```

```
}


typedef struct extent_s extent_t;
typedef void (*ext_fn)(extent_t *, void *, void *);
typedef struct extent_s
{
    void *idx;
    int rlen, vlen, ilen;

    ext_fn ext_fn;
    union {
        idx_fn idx_fn;
        fill_na_fn na_fn;
    };
    size_t vsz, rsz;
    extent_t *next;
} extent_t;

static void extent_access(extent_t *ext, void *r, void *v)
{
    ext->idx_fn(r, v, ext->idx, ext->rlen, ext->vlen, ext->ilen);
}

#define NEXT(p, q, start, end) ((p+q >= end) ? start : p+q)

static void extent_bool_get(extent_t *ext, void *r, void *v)
{
    rboolean_t *vi = ext->idx;
    rboolean_t *istart = vi, *iend = vi + ext->ilen;
    uint8_t *vv = v, *vr = r;
    size_t vsz = ext->vsz, rsz = ext->rsz;
    uint8_t *rstart = vr, *rend = vr + ext->rlen * rsz;

    for(int i=0; i<ext->vlen; i++)
    {
        rboolean_t b = *vi;

        vi = NEXT1(vi, istart, iend);
        if(b != false)
        {
            if(boolean_na(b))
                ext->na_fn(vr, rsz);
            else
                ext->next->ext_fn(ext->next, vr, vv);
            vr = NEXT(vr, rsz, rstart, rend);
        }
        vv += vsz;
    }
    if(ext->vlen < ext->ilen)
        ext->na_fn(vr, rend - vr);
}

static void extent_int_get(extent_t *ext, void *r, void *v)
{
    rint_t *vi = ext->idx;
    uint8_t *vv = v, *vr = r;
    size_t vsz = ext->vsz, rsz = ext->rsz;
```

```
    for(int i=0; i<ext->ilen; i++)
    {
        rint_t j = vi[i];
        if(int_na(j) || j > ext->vlen || j <= 0)
            ext->na_fn(vr, rsz);
        else
            ext->next->ext_fn(ext->next, vr, vv + (j-1) * vsz);
        vr += rsz;
    }
}

static void extent_missing_get_set(extent_t *ext, void *r, void *v)
{
    uint8_t *vv = v, *vr = r;
    size_t vsz = ext->vsz, rsz = ext->rsz;

    for(int i=0; i<ext->vlen; i++)
    {
        ext->next->ext_fn(ext->next, vr, vv);
        vr += rsz;
        vv += vsz;
    }
}

static void extent_bool_set(extent_t *ext, void *r, void *v)
{
    rboolean_t *vi = ext->idx;
    rboolean_t *istart = vi, *iend = vi + ext->ilen;
    uint8_t *vv = v, *vr = r;
    size_t vsz = ext->vsz, rsz = ext->rsz;

    for(int i=0; i<ext->vlen; i++)
    {
        rboolean_t b = *vi;
        vi = NEXT1(vi, istart, iend);
        if(b == true)
        {
            ext->next->ext_fn(ext->next, vr, vv);
            vr += rsz;
        }
        vv += vsz;
    }
}

static void extent_int_set(extent_t *ext, void *r, void *v)
{
    rint_t *vi = ext->idx;
    uint8_t *vv = v, *vr = r;
    size_t vsz = ext->vsz, rsz = ext->rsz;

    for(int i=0; i<ext->ilen; i++)
    {
        rint_t j = vi[i];
        if(!(int_na(j) || j > ext->vlen || j <= 0))
        {
            ext->next->ext_fn(ext->next, vr, vv + (j-1) * vsz);
            vr += rsz;
```

```
        }
    }
}

static ext_fn idx_ext_func(idx_kind kind, bool isset)
{
    switch(kind)
    {
    case SINGLE:
    case INDEX:
        return isset ? extent_int_set : extent_int_get;
    case MASK:
        return isset ? extent_bool_set : extent_bool_get;
    case MISSING:
        return extent_missing_get_set;
    }
    return NULL; /* NOTREACHED */
}

static inline int idx_dim(int vlen, robject_t *idx, idx_kind kind)
{
    switch(kind)
    {
    case SINGLE:
        return 1;
    case INDEX:
        return arith_length(idx);
    case MASK:
        return mask_result_len(vlen, arith_storage(idx), arith_length(idx));
    case MISSING:
        return vlen;
    }
    return 0; /* NOTREACHED */
}

static rtype_t **idx_types(rtype_t **ityps, robject_t *idx,
                           robject_t **idxs, int nidxs)
{
    ityps[0] = r_typeof(idx);
    for(int i=1; i < nidxs+1; i++)
        ityps[i] = r_typeof(idxs[i-1]);
    return ityps;
}

static int idx_extents(extent_t *exts, idx_kind *kinds,
                       robject_t *idx, robject_t **idxs, int nidxs,
                       size_t rsz, size_t vsz,
                       int *rshape, int *vshape,
                       int rcode, int vcode, bool isscal)
{
    extent_t *ext = &exts[0];
    extent_t *next = NULL;
    robject_t **pidx = &idx;
    int rlen = 1;
    int j = 0;
    int rdim;
    bool isset = (rcode != -1);
```

```
    for(int i=0; i<nidxs+1; ext++, i++)
    {
        int kind = kinds[i];

        rdim = idx_dim(vshape[i], *pidx, kind);
        if(kind != SINGLE)
            rshape[j++] = rdim;
        rlen *= rdim;

        *ext = (extent_t) {
            // scalar replacement: correct length for recycling in *_set_*_from_*
            .rlen = isscal ? 1 : rdim,
            .vlen = vshape[i],
            .ilen = arith_length(*pidx),
            .idx = arith_storage(*pidx),
        };

        if(i == 0)
        {
            ext->ext_fn = extent_access;
            ext->idx_fn = idx_access_func(kind, rcode, vcode);
        }
        else
        {
            ext->ext_fn = idx_ext_func(kind, isset);
            ext->na_fn = fill_na_funcs[vcode];
            // scalar replacement: do not advance replacement pointer in extent_set_*
            ext->rsz = isscal ? 0 : rsz;
            ext->vsz = vsz;
            ext->next = next;
        }
        rsz *= rdim;
        vsz *= vshape[i];
        next = ext;
        // pidx may get to &idx[nidxs], but the loop ends before it's dereferenced
        pidx = &idxs[i];
    }
    return rlen;
}

// copy dimnames from val (as indexed by exts) to res, if the latter is
// a vector or array
static void idx_extent_names(robject_t *res, robject_t *val, int rank,
                             extent_t *exts, idx_kind *kinds, int nexts,
                             int *rshape, int *vshape)
{
    rarray_t *arr = (rarray_t *)val;

    if(rank == 1)
    {
        // to element names
        rvector_t *rvec = (rvector_t *)res;
        rvec_add_names(rvec);

        // FIXME split into function
        rvector_t **dimnames = rvec_elts(arr->dimnames);

        for(int i=0; i<nexts; i++)
```

```
        {
            if(kinds[i] != SINGLE && dimnames[i])
            {
                // there will be exactly one of these
                idx_copy_names(rvec_elts(rvec->names), rvec_elts(dimnames[i]),
                               exts[i].idx, kinds[i], rshape[0], vshape[i],
                               exts[i].ilen);
            }
        }
    }
    else if(rank > 1)
    {
        // to dimnames
        rarray_t *rarr = (rarray_t *)res;
        rarr_add_names(rarr);

        // FIXME split into function
        rvector_t **srcnames = rvec_elts(arr->dimnames);
        rvector_t **destnames = rvec_elts(rarr->dimnames);
        int j = 0;

        for(int i=0; i<nexts; i++)
        {
            if(kinds[i] != SINGLE)
            {
                if(srcnames[i])
                {
                    destnames[j] = rvec_create(r_type_vec_symbol, rshape[j]);
                    idx_copy_names(rvec_elts(destnames[j]), rvec_elts(srcnames[i]),
                                   exts[i].idx, kinds[i], rshape[j], vshape[i],
                                   exts[i].ilen);
                }
                j++;
            }
        }
    }
}

bool check_arr_fetch(arr_chk_t *achk, rtype_t *typ, rtype_t **ityps,
                     int ntyps)
{
    int rank = 0;

    if(!rtype_is_array(typ))
        return false;
    for(int i=0; i<ntyps; i++)
    {
        if(!check_index(&achk->kinds[i], ityps[i]))
            return false;
        if(achk->kinds[i] != SINGLE)
            rank++;
    }
    achk->rank = rank;
    if(rank == 0) // single element
    {
        achk->typ = typ->elt;
        if(!rtype_is_scalar(typ->elt))
            achk->takeptr = true;
```

```
    }
    else if(rank == 1) // vector
        achk->typ = rvec_type_create(typ->elt);
    else // array
        achk->typ = typ;
    return true;
}

robject_t *arr_fetch(vm_ctx_t *vm, robject_t *val, robject_t *idx,
                     robject_t **idxs, int nidxs)
{
    rtype_t *typ = r_typeof(val);
    robject_t *res = NULL;
    arr_chk_t achk = { .kinds = alloca((nidxs+1) * sizeof(idx_kind)) };
    rtype_t **ityps = idx_types(alloca((nidxs+1) * sizeof(rtype_t *)),
                                idx, idxs, nidxs);

    if(!check_arr_fetch(&achk, typ, ityps, nidxs+1))
        vm_error(vm, "[", "invalid argument type.");

    rarray_t *arr = (rarray_t *)val;
    int vrank = arr->rank;
    int *vshape = arr->shape;

    if(vrank > nidxs+1)
        vm_error(vm, "[", "too few indices.");
    else if(vrank < nidxs+1)
        vm_error(vm, "[", "too many indices.");

    rtype_t *rtyp = vm_retain(vm, achk.typ);
    int rrank = achk.rank;
    int *rshape = alloca(rrank * sizeof(int));

    extent_t *exts = alloca((nidxs+1) * sizeof(extent_t));
    int rlen = idx_extents(exts, achk.kinds, idx, idxs, nidxs,
                           rtype_eltsz(typ->elt), rtype_eltsz(typ->elt),
                           rshape, vshape,
                           -1, idx_func_code(typ->elt), false);

    void *r = NULL;
    void *v = arith_storage(val);

    if(achk.takeptr)
        r = &res;
    else
    {
        res = arith_create(rtyp, rlen);
        r = arith_storage(res);
        if(rrank > 1)
            rarr_set_shape((rarray_t *)res, rrank, rshape);
    }
    vm_retain(vm, res);
    if(rlen > 0)
    {
        extent_t *ext = &exts[nidxs];
        ext->ext_fn(ext, r, v);
    }
    // copy dimnames
```

```
    if(is_named(val))
        idx_extent_names(res, val, rrank, exts, achk.kinds,
                         nidxs+1, rshape, vshape);
    vm_release(vm, 2);
    return res;
}

bool check_arr_store(arr_chk_t *achk, rtype_t *typ, rtype_t **ityps,
                     int ntyps, rtype_t *rtyp)
{
    rtype_t *ret, *et;
    int rank = 0;

    if(!rtype_is_array(typ))
        return false;
    for(int i=0; i<ntyps; i++)
    {
        if(!check_index(&achk->kinds[i], ityps[i]))
            return false;
        if(achk->kinds[i] != SINGLE)
            rank++;
    }
    achk->rank = rank;
    if(rtype_is_container(rtyp) && rank > 0)
    {
        // replace with contents: check against element type
        ret = rtyp->elt;
    }
    else // either replacement not a container, or single value required
    {
        // replace with value
        ret = rtyp;
        // replace with the reference as a value
        if(!rtype_is_scalar(rtyp))
            achk->takeptr = true;
    }
    // container is always an array
    et = typ->elt;
    // element types
    if(rtype_is_scalar(et))
    {
        // can't accept reference objects
        if(!rtype_is_scalar(ret))
            return false;
    }
    else
    {
        // can't accept incompatible objects
        if(!r_subtypep(ret, et))
            return false;
    }
    achk->typ = ret;
    return true;
}

robject_t *arr_store(vm_ctx_t *vm, robject_t *val, robject_t *rpl,
                     robject_t *idx, robject_t **idxs, int nidxs)
{
```

```
    rtype_t *typ = r_typeof(val), *rtyp = r_typeof(rpl);
    arr_chk_t achk = { .kinds = alloca((nidxs+1) * sizeof(idx_kind)) };
    rtype_t **ityps = idx_types(alloca((nidxs+1) * sizeof(rtype_t *)),
                                idx, idxs, nidxs);

    if(!check_arr_store(&achk, typ, ityps, nidxs+1, rtyp))
        vm_error(vm, "[=", "invalid argument type.");

    rarray_t *arr = (rarray_t *)val;
    int vrank = arr->rank;
    int *vshape = arr->shape;

    if(vrank > nidxs+1)
        vm_error(vm, "[=", "too few indices.");
    else if(vrank < nidxs+1)
        vm_error(vm, "[=", "too many indices.");

    // 'requirements' that rpl must conform to
    int rrank = achk.rank;
    int *rshape = alloca(rrank * sizeof(int));

    extent_t *exts = alloca((nidxs+1) * sizeof(extent_t));
    int rlen = idx_extents(exts, achk.kinds, idx, idxs, nidxs,
                           rtype_eltsz(achk.typ), rtype_eltsz(typ->elt),
                           rshape, vshape,
                           idx_func_code(achk.typ), idx_func_code(typ->elt),
                           rtype_is_scalar(rtyp));

    if(rtype_is_vector(rtyp))
    {
        rvector_t *rvec = (rvector_t *)rpl;
        if(rvec_len(rvec) < rlen)
            vm_error(vm, "[=", "non-conforming replacement vector.");
    }
    else if(rtype_is_array(rtyp))
    {
        rarray_t *rarr = (rarray_t *)rpl;
        if(!rarr_shape_conform(rrank, rarr->rank, rshape, rarr->shape))
            vm_error(vm, "[=", "non-conforming replacement array.");
    }

    void *r = NULL;
    void *v = arith_storage(val);

    if(achk.takeptr)
        r = &rpl;
    else
        r = arith_storage(rpl);

    if(rlen > 0)
    {
        extent_t *ext = &exts[nidxs];
        ext->ext_fn(ext, r, v);
    }
    return rpl;
}
//}}
```

arith/builtins.c

⟨*arith/builtins.c*⟩≡

```
#include "global.h"
#include "ir.h"
#include "rt/builtin.h"
#include "vm/vm_ops.h"
#include "arith/scalar.h"
#include "arith/vec.h"
#include "arith/dispatch.h"

static void builtin_binary(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x, *y;
    } end_args(args, a);
    const arith_builtin_t *ab = fn->cbi->data;
    robject_t *res = arith_binary(vm, ab->fntab, ab->is_pred, a->x, a->y);
    builtin_return(r, robject_t *, res);
}

static void builtin_unary(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x;
    } end_args(args, a);
    const arith_builtin_t *ab = fn->cbi->data;
    robject_t *res = arith_unary(vm, ab->fntab, ab->is_pred, a->x);
    builtin_return(r, robject_t *, res);
}

static void builtin_reduce(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x;
    } end_args(args, a);
    const arith_builtin_t *ab = fn->cbi->data;
    robject_t *res = arith_reduce(vm, ab->fntab, a->x);
    builtin_return(r, robject_t *, res);
}

static void builtin_convert(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x;
    } end_args(args, a);
    rtype_t *etyp = *(rtype_t **)fn->cbi->data; // note: not an arith_builtin_t
    robject_t *res = arith_convert(vm, a->x, etyp);
    builtin_return(r, robject_t *, res);
}

static void builtin_coerce(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x;
        rtype_t *etyp;
    } end_args(args, a);
    if(!rtype_is_scalar(a->etyp))
```

```
        vm_error(vm, "as_type", "invalid argument type.");
    robject_t *res = arith_convert(vm, a->x, a->etyp);
    builtin_return(r, robject_t *, res);
}

static inline bool fill_scal(rbuf_t *dest, robject_t *val, rtype_t *etyp, rtype_t *typ)
{
    rvalue_union_t temp;
    size_t esz = rtype_eltsz(etyp);
    int len = dest->length;
    uint8_t *ptr = dest->elts;

    // arithmetic conversion if needed
    if(typ == etyp)
        r_unbox(&temp, val);
    else if(!rtype_is_scalar(typ) || !scalar_convert(&temp, val, etyp, typ))
        return false;
    for(int i=0; i<len; i++, ptr+=esz)
        memcpy(ptr, &temp, esz);
    return true;
}

static inline bool fill_ptr(rbuf_t *dest, robject_t *val, rtype_t *etyp, rtype_t *typ)
{
    robject_t **ptr = dest->elts;
    int len = dest->length;

    if(!r_subtypep(typ, etyp))
        return false;
    for(int i=0; i<len; i++, ptr++)
        *ptr = val;
    return true;
}

static inline bool fill_buf(rbuf_t *dest, robject_t *val)
{
    rtype_t *etyp = elt_type(dest);
    rtype_t *typ = r_typeof(val);

    if(rtype_is_scalar(etyp))
        return fill_scal(dest, val, etyp, typ);
    return fill_ptr(dest, val, etyp, typ);
}

static inline void fill_buf_from(rbuf_t *dest, rbuf_t *src)
{
    copy_recycle(dest->elts, src->elts, dest->length, src->length,
                 rtype_eltsz(elt_type(dest)));
}

// create a vector of given element type and length, retaining both
// (i.e. +2 release)
static rvector_t *safe_vec_create(vm_ctx_t *vm, rtype_t *etyp, unsigned length)
{
    rtype_t *vtyp = vm_retain(vm, rvec_type_create(etyp));
    return vm_retain(vm, rvec_create(vtyp, length));
}
```

```
static rarray_t *safe_arr_create(vm_ctx_t *vm, rtype_t *etyp, int rank, int *dims)
{
    rtype_t *atyp = vm_retain(vm, rarr_type_create(etyp));
    return vm_retain(vm, rarr_create(atyp, rank, dims));
}

// extract the types of each pointer in elts[nelts]
// return the least(ish!) common supertype of them all
static rtype_t *examine_args(void **elts, int nelts)
{
    rtype_t *etyp = NULL;
    for(int i=0; i<nelts; i++)
    {
        rtype_t *typ = r_typeof(elts[i]);
        if(!etyp)
            etyp = typ;
        else
            etyp = r_common_type(typ, etyp);
    }
    return etyp ? etyp : r_type_object;
}

// mimicking R would mean recursive descent into member vectors during
// element type determination and copying. we don't do this.

// creates a new vector containing the elements of the rest vector.
// if they're all the same type (or are scalars so can be converted),
// the result has that element type.
static void builtin_vec(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rvector_t *rest;
    } end_args(args, a);

    if(!a->rest)
    {
        builtin_return(r, robject_t *, NULL);
        return;
    }

    rvector_t *src = a->rest;
    rtype_t *etyp = examine_args(rvec_elts(src), rvec_len(src));
    // XXX move to arith.c??
    if(rtype_is_scalar(etyp))
    {
        robject_t *res = arith_convert(vm, (robject_t *)src, etyp);
        builtin_return(r, robject_t *, res);
    }
    else
    {
        rvector_t *dest = safe_vec_create(vm, etyp, rvec_len(src));
        memcpy(rvec_elts(dest), rvec_elts(src), rvec_len(src) * sizeof(robject_t *));
        // XXX don't bother, rest list construction does the wrong thing
        // with named args, due to two-phase arg matching
        if(is_named((robject_t *)src))
            copy_names((robject_t *)dest, (robject_t *)src);
        builtin_return(r, rvector_t *, dest);
        vm_release(vm, 2);
```

```
    }
}

static rvector_t *typed_vec_init(vm_ctx_t *vm, rtype_t *etyp, int len, robject_t *val)
{
    rvector_t *vec = safe_vec_create(vm, etyp, len);

    if(!val)
        memset(rvec_elts(vec), 0, len * rtype_eltsz(etyp));
    else if(!fill_buf(&vec->buf, val))
        vm_error(vm, "vec_create", "invalid argument type.");
    vm_release(vm, 2);
    return vec;
}

// XXX rationalise these; we want a minimal, orthogonal, useful set of
// constructors
static void builtin_typed_vec(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        rint_t len;
        robject_t *val; // optional
    } end_args(args, a);
    argbits_t ab = a->argbits; // TEMP

    if(a->len < 0 || a->len == rint_na)
        vm_error(vm, "typed_vec", "invalid argument.");

    rtype_t *etyp = *(rtype_t **)fn->cbi->data;
    rvector_t *res = typed_vec_init(vm, etyp, a->len, !missingp(ab, 1) ? a->val : NULL);
    builtin_return(r, rvector_t *, res);
}

static void builtin_vector(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        rtype_t *etyp;
        rint_t len;
        robject_t *val; // optional
    } end_args(args, a);
    argbits_t ab = a->argbits; // TEMP

    if(a->len < 0 || a->len == rint_na)
        vm_error(vm, "vector", "invalid argument.");

    rvector_t *res = typed_vec_init(vm, a->etyp, a->len, !missingp(ab, 2) ? a->val : NULL);
    builtin_return(r, rvector_t *, res);
}

// mimicks some R behaviour:
// length(anything except a container) = 1
// length(NULL) = 0
static void builtin_length(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x;
```

```
    } end_args(args, a);
    rint_t len;
    if(a->x)
    {
        rtype_t *typ = r_typeof(a->x);
        if(rtype_is_container(typ))
            len = ((rbuf_t *)a->x)->length;
        else
            len = 1;
    }
    else
        len = 0;
    builtin_return(r, rint_t, len);
}

static void builtin_names_get(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rvector_t *x;
    } end_args(args, a);
    // names(nil) is nil
    rvector_t *res = a->x ? a->x->names : NULL;
    builtin_return(r, rvector_t *, res);
}

static void builtin_names_set(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rvector_t *x;
        rvector_t *names;
    } end_args(args, a);

    if(!a->x)
        vm_error(vm, "names=", "invalid argument.");
    if(a->names)
    {
        assert(r_typeof(a->names) == r_type_vec_symbol);
        if(rvec_len(a->names) != rvec_len(a->x))
            vm_error(vm, "names=", "invalid argument length.");
    }
    a->x->names = a->names;
    builtin_return(r, rvector_t *, a->names);
}

static rvector_t *copy_dimnames(vm_ctx_t *vm, rvector_t *src)
{
    int len = rvec_len(src);
    rvector_t **srcnames = rvec_elts(src);
    rvector_t *dest = safe_vec_create(vm, r_type_vec_symbol, len);
    rvector_t **destnames = rvec_elts(dest);

    memcpy(destnames, srcnames, sizeof(rvector_t *) * len);
    vm_release(vm, 2);
    return dest;
}

static void builtin_dimnames_get(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
```

```
    def_args {
        rarray_t *x;
    } end_args(args, a);
    // mutation could break the shape invariant! must take a copy.
    rvector_t *res = NULL;
    if(a->x && a->x->dimnames)
        res = copy_dimnames(vm, a->x->dimnames);
    builtin_return(r, rvector_t *, res);
}

static void builtin_dimnames_set(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rarray_t *x;
        rvector_t *dimnames;
    } end_args(args, a);

    if(!a->x)
        vm_error(vm, "dimnames=", "invalid argument.");
    if(a->dimnames)
    {
        if(r_typeof(a->dimnames) != r_type_vec_names)
            vm_error(vm, "dimnames=", "invalid argument type.");
        if(rvec_len(a->dimnames) != a->x->rank)
            vm_error(vm, "dimnames=", "invalid argument length.");

        rvector_t **dimnames = rvec_elts(a->dimnames);
        for(int i=0; i<a->x->rank; i++)
        {
            if(dimnames[i])
            {
                if(r_typeof(dimnames[i]) != r_type_vec_symbol)
                    vm_error(vm, "dimnames=", "invalid argument type.");
                if(rvec_len(dimnames[i]) != a->x->shape[i])
                    vm_error(vm, "dimnames=", "invalid argument length.");
            }
        }
    }
    a->x->dimnames = copy_dimnames(vm, a->dimnames);
    builtin_return(r, rvector_t *, a->dimnames);
}

static void builtin_fetch(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        robject_t *obj;
        robject_t *idx; // optional
        rvector_t *rest; // optional
    } end_args(args, a);
    robject_t *res;

    // not an R-ism: nil[anything] is an error
    if(!a->obj)
        vm_error(vm, "[", "invalid subscript of nil.");
    if(!a->rest)
        res = vec_fetch(vm, a->obj, a->idx);
    else
```

```
        res = arr_fetch(vm, a->obj, a->idx,
                        rvec_elts(a->rest), rvec_len(a->rest));
    builtin_return(r, robject_t *, res);
}

static void builtin_store(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        robject_t *obj;
        robject_t *val;
        robject_t *idx; // optional
        rvector_t *rest; // optional
    } end_args(args, a);
    robject_t *res;

    // R-ism: nil[anything] = anything is an error
    if(!a->obj)
        vm_error(vm, "[=", "invalid assignment to nil.");
    if(!a->rest)
        res = vec_store(vm, a->obj, a->val, a->idx);
    else
        res = arr_store(vm, a->obj, a->val, a->idx,
                        rvec_elts(a->rest), rvec_len(a->rest));
    builtin_return(r, robject_t *, res);
}

// XXX this is not very useful.
static void builtin_vec_cat(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rvector_t *rest;
    } end_args(args, a);
    int len = 0;
    rvector_t *rest = a->rest;
    rtype_t *etyp = NULL;

    if(!rest)
    {
        builtin_return(r, robject_t *, NULL);
        return;
    }

    rvector_t **vecs = rvec_elts(rest);
    int nrest = rvec_len(rest);

    for(int i=0; i < nrest; i++)
    {
        if(!rtype_is_vector(r_typeof(vecs[i])))
            vm_error(vm, "cat", "can't cat things which aren't vectors.");
        if(i == 0)
            etyp = elt_type(&vecs[i]->buf);
        else if(etyp != elt_type(&vecs[i]->buf))
            vm_error(vm, "cat", "all arguments must be of the same type.");
        len += rvec_len(vecs[i]);
    }

    rvector_t *dest = safe_vec_create(vm, etyp, len);
```

```
    uint8_t *destptr = rvec_elts(dest);

    for(int i=0; i < nrest; i++)
    {
        int alen = rvec_len(vecs[i]);
        size_t sz = alen * rtype_eltsz(etyp);

        memcpy(destptr, rvec_elts(vecs[i]), sz);
        destptr += sz;
    }

    // XXX concatenate names
    builtin_return(r, rvector_t *, dest);
    vm_release(vm, 2);
}

static void builtin_runif(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        rint_t n;
        rdouble_t min, max; // both optional
    } end_args(args, a);
    argbits_t ab = a->argbits; // TEMP

    if(isnan(a->min) || isnan(a->max) || a->n < 0
       || a->n == rint_na || a->max < a->min)
        vm_error(vm, "runif", "invalid argument.");

    rvector_t *res = safe_vec_create(vm, r_type_double, a->n);
    rdouble_t *vals = rvec_elts(res);

    if(missingp(ab, 1))
        a->min = 0;
    if(missingp(ab, 2))
        a->max = 1;

    rdouble_t span = a->max - a->min;
    for(int i=0; i<a->n; i++)
        vals[i] = unif_rand() * span + a->min;
    builtin_return(r, rvector_t *, res);
    vm_release(vm, 2);
}

static void builtin_rnorm(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        rint_t n;
        rdouble_t mu, sigma; // both optional
    } end_args(args, a);
    argbits_t ab = a->argbits; // TEMP

    if(a->n < 0 || a->n == rint_na)
        vm_error(vm, "rnorm", "invalid argument.");

    rvector_t *res = safe_vec_create(vm, r_type_double, a->n);
    rdouble_t *vals = rvec_elts(res);
```

```
    if(missingp(ab, 1))
        a->mu = 0;
    if(missingp(ab, 2))
        a->sigma = 1;
    for(int i=0; i<a->n; i++)
        vals[i] = rnorm(a->mu, a->sigma);
    builtin_return(r, rvector_t *, res);
    vm_release(vm, 2);
}


static void builtin_range(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        rint_t from, to;
    } end_args(args, a);

    if(a->from == rint_na || a->to == rint_na)
        vm_error(vm, ":", "invalid argument.");

    int step = (a->from <= a->to) ? 1 : -1;
    int len = abs(a->to - a->from) + 1;
    rvector_t *res = safe_vec_create(vm, r_type_int, len);
    rint_t *vals = rvec_elts(res);

    for(int i = a->from; i != a->to + step; i += step, vals++)
        *vals = i;

    builtin_return(r, rvector_t *, res);
    vm_release(vm, 2);
}

static rarray_t *typed_arr_init(vm_ctx_t *vm, rtype_t *etyp, int rank,
                                int *dims, robject_t *val)
{
    rarray_t *arr = safe_arr_create(vm, etyp, rank, dims);
    rtype_t *typ = r_typeof(val);

    // all-bits-zero by default
    if(!val)
        memset(rvec_elts(arr), 0, rvec_len(arr) * rtype_eltsz(etyp));
    // vector value and its elements are the right type; recycle along length if needed
    else if(rtype_is_container(typ) && typ->elt == etyp)
        fill_buf_from(&arr->buf, (rbuf_t *)val);
    // single value, initialise all cells with it
    else if(!fill_buf(&arr->buf, val))
        vm_error(vm, "array", "invalid argument type.");
    vm_release(vm, 2);
    return arr;
}

static void builtin_array(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        argbits_t argbits;
        rtype_t *etyp;
        rvector_t *dims;
```

```
        robject_t *val; // optional
    } end_args(args, a);
    argbits_t ab = a->argbits; // TEMP

    if(!a->dims || !a->etyp)
        vm_error(vm, "array", "invalid argument.");

    // rank is length(dims)
    int rank = rvec_len(a->dims),
        *dims = rvec_elts(a->dims);

    // elements must be nonnegative and not NA
    for(int i=0; i < rank; i++)
        if(dims[i] < 0 || dims[i] == rint_na)
            vm_error(vm, "array", "invalid dimension.");

    rarray_t *res = typed_arr_init(vm, a->etyp, rank, dims,
                                   !missingp(ab, 2) ? a->val : NULL);
    builtin_return(r, rarray_t *, res);
}

static void builtin_shape(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x;
    } end_args(args, a);

    rtype_t *type = r_typeof(a->x);
    rvector_t *res = NULL;

    if(rtype_is_array(type))
    {
        rarray_t *arr = (rarray_t *)a->x;

        res = safe_vec_create(vm, r_type_int, arr->rank);
        memcpy(rvec_elts(res), arr->shape, arr->rank * sizeof(int));
        vm_release(vm, 2);
    }
    else
    {
        int len;

        if(rtype_is_vector(type))
            len = ((rbuf_t *)a->x)->length;
        else
            len = 1;
        res = safe_vec_create(vm, r_type_int, 1);
        *(int *)rvec_elts(res) = len;
        vm_release(vm, 2);
    }
    builtin_return(r, rvector_t*, res);
}

static void builtin_rank(vm_ctx_t *vm, rbuiltin_t *fn, uint8_t *args, void *r)
{
    def_args {
        robject_t *x;
    } end_args(args, a);
```

```
    rtype_t *type = r_typeof(a->x);
    rint_t rank;

    if(rtype_is_array(type))
        rank = ((rarray_t *)a->x)->rank;
    else if(rtype_is_vector(type))
        rank = 1;
    else
        rank = 0;
    builtin_return(r, rint_t, rank);
}


const static builtin_init_arg_t binary_args[] =
    init_args({ "x", &r_type_object },
              { "y", &r_type_object });
const static builtin_init_arg_t atan2_args[] =
    init_args({ "y", &r_type_object },
              { "x", &r_type_object });
const static builtin_init_arg_t unary_args[] =
    init_args({ "x", &r_type_object });
const static builtin_init_arg_t create_args[] =
    init_args({ "len", &r_type_int, false },
              { "val", &r_type_object, true });

const extern builtin_ops_t binary_ops, unary_ops, binary_boolean_ops,
    unary_boolean_ops, conv_ops, reduce_ops, fetch_ops, store_ops,
    vector_ops, array_ops, pure_ops;

#define init_arith(ft, p, c) \
    &(arith_builtin_t) { .fntab = ft, .is_pred = p, .op = c }

#define _defarith(n, f, a, o, ft, p, c) \
    { init_builtin(n, f, &r_type_object, a), \
      .cbi = init_cbi(o, n, init_arith(ft, p, c)) }

#define defoper(name, op, pred, opcode, n, k...) \
    _defarith(name, builtin_##n##ary,  n##ary_args, &n##ary##k##_ops, \
              PASTE2(op, _funcs), pred, opcode)

// these have cbuiltins for type recovery, but don't generate VM instructions
#define defbinary(name, op, pred) defoper(name, op, pred, 0, bin)
#define defunary(name, op, pred) defoper(name, op, pred, 0, un)
// these have instructions to go with them
#define defbin_ins(name, op, pred) defoper(name, op, pred, ENUM(op), bin)
#define defuna_ins(name, op, pred) defoper(name, op, pred, ENUM(op), un)
// these ones are never predicates, but need a different .cbi.ops
#define defbin_bool(name, op) defoper(name, op, false, ENUM(op, boolean), bin, _boolean)
#define defuna_bool(name, op) defoper(name, op, false, ENUM(op, boolean), un, _boolean)

// the function and the reduction operator have different names
#define defreduce(name, op, pred) \
    _defarith(name, builtin_reduce, unary_args, &reduce_ops, \
              PASTE3(reduce_, op, _funcs), pred, 0)

// .cbi is needed for these; the builtin_ function looks at .cbi.data to
// know what type it was called at
```

```
// converts its input to have element type t
#define defconv(n, t) \
    { init_builtin(n, builtin_convert, &r_type_object, unary_args), \
      .cbi = init_cbi(&conv_ops, n, &RTYPE(t)) }
// makes a new vector of type t
#define defcreate(n, t) \
    { init_builtin(n, builtin_typed_vec, &r_type_vec_##t, create_args), \
      .cbi = init_cbi(NULL, n, &RTYPE(t)) }
// FIXME: generate these from the m4 _ops lists
const builtin_init_t arith_builtins[] = {
    defbin_ins("+", add, false),
    defbin_ins("-", sub, false),
    defbin_ins("*", mul, false),
    defbin_ins("/", div, false),

    defbin_bool("&", and),
    defbin_bool("|", or),

    defbin_ins("==", eql, true),
    defbin_ins("!=", neq, true),
    defbin_ins("<", lth, true),
    defbin_ins("<=", lte, true),
    defbin_ins(">", gth, true),
    defbin_ins(">=", gte, true),

    defbin_ins("^", pow, false),
    // because the arguments need to be named "y", "x"...
    _defarith("atan2", builtin_binary,  atan2_args,
              &binary_ops, atan2_funcs, false, 0),

    defuna_bool("!", not),
    defuna_ins("0-", neg, false),

    defuna_ins("is_na", is_na, true),

    defunary("sqrt", sqrt, false),
    defunary("sin", sin, false),
    defunary("cos", cos, false),
    defunary("tan", tan, false),
    defunary("asin", asin, false),
    defunary("acos", acos, false),
    defunary("atan", atan, false),
    defunary("log", log, false),
    defunary("exp", exp, false),
    defunary("round", round, false),

    defreduce("any", or, true),
    defreduce("all", and, true),
    defreduce("sum", add, false),
    defreduce("prod", mul, false),

    defconv("as_bool", boolean),
    defconv("as_int", int),
    defconv("as_double", double),

    defcreate("bool", boolean),
    defcreate("int", int),
```

```
defcreate("dbl", double),
defcreate("list", object), // slight abuse of notation

defbuiltin("as_type", builtin_coerce, &r_type_object,
           { "x", &r_type_object },
           { "type", &r_type_type }),
defbuiltin("vec", builtin_vec, &r_type_vector,
           { "...", &r_type_vec_object, true }),
defcbuiltin("vector", builtin_vector, &r_type_vector,
            &vector_ops, NULL,
           { "etyp", &r_type_type, false },
           { "len", &r_type_int, false },
           { "val", &r_type_object, true }),
defcbuiltin("length", builtin_length, &r_type_int,
            &pure_ops, NULL,
           { "x", &r_type_object, false }),
defbuiltin("names", builtin_names_get, &r_type_vec_symbol,
           { "x", &r_type_vector, false }),
defbuiltin("names=", builtin_names_set, &r_type_vec_symbol,
           { "x", &r_type_vector, false },
           { "names", &r_type_vec_symbol, false }),
defbuiltin("dimnames", builtin_dimnames_get, &r_type_vec_names,
           { "x", &r_type_array, false }),
defbuiltin("dimnames=", builtin_dimnames_set, &r_type_vec_names,
           { "x", &r_type_array, false },
           { "dimnames", &r_type_vec_names, false }),
defbuiltin("cat", builtin_vec_cat, &r_type_vector,
           { "...", &r_type_vec_object, true }),
defbuiltin("runif", builtin_runif, &r_type_vec_double,
           { "n", &r_type_int, false },
           { "min", &r_type_double, true },
           { "max", &r_type_double, true }),
defbuiltin("rnorm", builtin_rnorm, &r_type_vec_double,
           { "n", &r_type_int, false },
           { "mu", &r_type_double, true },
           { "sigma", &r_type_double, true }),
defcbuiltin("array", builtin_array, &r_type_array,
            &array_ops, NULL,
           { "etyp", &r_type_type, false },
           { "dims", &r_type_vec_int, false },
           { "val", &r_type_object, true }),
defbuiltin("shape", builtin_shape, &r_type_vec_int,
           { "x", &r_type_object, false }),
defcbuiltin("rank", builtin_rank, &r_type_int,
            &pure_ops, NULL,
           { "x", &r_type_object, false }),
defbuiltin(":", builtin_range, &r_type_vec_int,
           { "from", &r_type_int, false },
           { "to", &r_type_int, false }),
defcbuiltin("[", builtin_fetch, &r_type_object,
            &fetch_ops, NULL,
           { "vec", &r_type_object, false },
           { "i", &r_type_object, true },
           { "...", &r_type_vec_object, true }),
defcbuiltin("[=", builtin_store, &r_type_object,
            &store_ops, NULL,
           { "vec", &r_type_object, false },
           { "val", &r_type_object, false },
```

```
                { "i", &r_type_object, true },
                { "...", &r_type_vec_object, true }),
    { }
};
```

arith/opt_builtins.c

⟨*arith/opt_builtins.c*⟩≡

```
#include "global.h"
#include "ir.h"
#include "opt.h"
#include "vm/vm_ops.h"
#include "gen_code.h"
#include "arith/scalar.h"
#include "arith/vec.h"
#include "arith/dispatch.h"

static void gen_arith(cnode_t *node)
{
    const arith_builtin_t *ab = node->builtin.bi->data;
    assert(ab->op);
    // OP.type dest, args...
    asm_op_type(ab->op, node->builtin.optype);
    asm_stack(loc_for(node));
    asm_args(&node->builtin.args);
}

static void gen_generic(cnode_t *node)
{
    const arith_builtin_t *ab = node->builtin.bi->data;
    assert(ab->op);
    // OP dest, args...
    asm_op(ab->op);
    asm_stack(loc_for(node));
    asm_args(&node->builtin.args);
}

#if 0
// XXX unused!
static void gen_width(cnode_t *node)
{
    assert(!"handled");
    op_code_t op = (op_code_t)(intptr_t)node->builtin.bi->data;

    // OP.width dest, obj, idx
    asm_op_width(op, node->builtin.optype);
    asm_stack(loc_for(node));
    asm_args(&node->builtin.args);
}
#endif

// arith
static inline cresult
fold_binary(binop_chk_t *chk, binop_fn fn, cell_t *cell, cell_t *xc, cell_t *yc)
{
    // constant values available?
    if(opt.opt_constfold && fn && cell_const_obj(xc) && cell_const_obj(yc))
```

```
    {
        robject_t *xv = cell_const(xc), *yv = cell_const(yc);
        rvalue_union_t val;
        robject_t *obj;

        // fold operation on constants
        fn(&val, BOXPTR(xv), BOXPTR(yv), 1);
        obj = c_intern(r_box(chk->typ, &val));
        cell_set_const(cell, obj); // sets type
        return CHANGED;
    }
    return SUCCESS;
}

static cresult trans_binary(opt_phase phase, cnode_t *node, cell_t *cell,
                            const cbuiltin_t *bi, cnode_array_t *args)
{
    const arith_builtin_t *ab = bi->data;

    if(!check_nargs(2, node, args))
        return SUCCESS;

    cell_t *xc = cell_for(aref(args, 0));
    cell_t *yc = cell_for(aref(args, 1));
    rtype_t *xt = cell_type(xc), *yt = cell_type(yc);

    // operand types available?
    if(!xt || !yt)
        return SUCCESS;

    binop_chk_t chk = { 0 };

    // operation is valid on these types?
    if(!check_binary(&chk, xt, yt, ab->is_pred))
    {
        // check_ requires precise types; this call may not be
        // statically invalid, but we can't tell, so punt to runtime
        return SUCCESS;
    }

    // transfer the return type if required
    if(phase == TRANSFER)
        cell_set_type(cell, chk.typ);

    // scalars only
    // XXX allow constant-folding on nonscalars, but never become_builtin
    if(chk.xvec || chk.yvec)
        return SUCCESS;

    binop_fn fn = dispatch_binary(&chk, ab->fntab);

    if(phase == TRANSFER)
    {
        // constant values available?
        return fold_binary(&chk, fn, cell, xc, yc);
    }
    if(!fn) // else phase == TRANSFORM
    {
```

```
        c_error("operation `%s` not supported on `%s` and `%s`.",
                bi->name, rtype_name(xt), rtype_name(yt));
        return FAILED;
    }
    if(ab->op)
    {
        // lower the CALL if possible
        call_become_builtin(node, bi, chk.optyp, chk.typ);
        return CHANGED;
    }
    return SUCCESS;
}

static cresult trans_unary(opt_phase phase, cnode_t *node, cell_t *cell,
                           const cbuiltin_t *bi, cnode_array_t *args)
{
    const arith_builtin_t *ab = bi->data;

    if(!check_nargs(1, node, args))
        return SUCCESS;

    cell_t *xc = cell_for(aref(args, 0));
    rtype_t *xt = cell_type(xc);

    if(!xt)
        return SUCCESS;

    unop_chk_t chk = { 0 };

    if(!check_unary(&chk, xt, ab->is_pred))
        return SUCCESS;
    if(phase == TRANSFER)
        cell_set_type(cell, chk.typ);
    if(chk.vec)
        return SUCCESS;

    unop_fn fn = dispatch_unary(&chk, ab->fntab);

    if(phase == TRANSFER)
    {
        if(opt.opt_constfold && fn && cell_const_obj(xc))
        {
            robject_t *xv = cell_const(xc);
            rvalue_union_t val;
            robject_t *obj;

            fn(&val, BOXPTR(xv), 1);
            obj = c_intern(r_box(chk.typ, &val));
            cell_set_const(cell, obj);
            return CHANGED;
        }
        return SUCCESS;
    }
    if(!fn) // else phase == TRANSFORM
    {
        c_error("operation `%s` not supported on `%s`.",
                bi->name, rtype_name(xt));
        return FAILED;
```

```
    }
    if(ab->op)
    {
        assert(!chk.vec);
        call_become_builtin(node, bi, xt, chk.typ); // operation occurs at argument type, for
        return CHANGED;
    }
    return SUCCESS;
}

static cresult trans_reduce(opt_phase phase, cnode_t *node, cell_t *cell,
                            const cbuiltin_t *bi, cnode_array_t *args)
{
    if(!check_nargs(1, node, args))
        return SUCCESS;

    rtype_t *xt = arg_type(args, 0);

    if(!xt)
        return SUCCESS;

    reduce_chk_t chk = { 0 };

    if(check_reduce(&chk, xt) && phase == TRANSFER)
        cell_set_type(cell, chk.typ);
    return SUCCESS;
}


static cresult enforce_arith(cnode_t *node)
{
    cnode_array_t *args = &node->builtin.args;
    cresult res = SUCCESS;
    int i;

    assert(node->builtin.optype);
    // all args coerced to the given type
    array_foreach(args, i)
    {
        cnode_t **ptr = aptr(args, i);
        res |= enforce_decl(node, node->builtin.optype, ptr, true);
    }
    return res;
}

// if an alias may be created because of this answer, only scalars
// (i.e. lowered to BUILTIN) are safe; otherwise we're fine.
static bool arith_is_pure(cnode_t *node, bool may_alias)
{
    return !may_alias || node->type == CN_BUILTIN;
}

// if an alias may be created because of this answer, the answer is no
static bool cons_is_pure(cnode_t *node, bool may_alias)
{
    return !may_alias;
}
```

```
const builtin_ops_t
    binary_ops = { trans_binary, enforce_arith, gen_arith,
                   .is_pure = arith_is_pure },
    unary_ops = { trans_unary, enforce_arith, gen_arith,
                  .is_pure = arith_is_pure },
    binary_boolean_ops = { trans_binary, enforce_arith, gen_generic,
                           .is_pure = arith_is_pure },
    unary_boolean_ops = { trans_unary, enforce_arith, gen_generic,
                          .is_pure = arith_is_pure },
    reduce_ops = { trans_reduce, NULL, NULL };

#if 0
// conv, FIXME
static cresult trans_conv(opt_phase phase, cnode_t *node, cell_t *cell,
                          const cbuiltin_t *bi, cnode_array_t *args)
{
    // return become_copy(node, ...);
    return SUCCESS;
}
#endif
const builtin_ops_t conv_ops = { };

// container fetch and store
static rtype_t *trans_vec_fetch(opt_phase phase, cnode_array_t *args)
{
    rtype_t *vt = arg_type(args, 0), *it = arg_type(args, 1);

    if(!vt || !it)
        return NULL;

    idx_chk_t chk = { 0 };

    if(!check_fetch(&chk, vt, it))
        return NULL;
    if(phase == TRANSFER)
        return chk.typ;
    return (chk.kind == SINGLE) ? chk.typ : NULL;
}

static rtype_t *trans_arr_fetch(opt_phase phase, cnode_array_t *args)
{
    int rank = alen(args) - 1;
    rtype_t *vt = arg_type(args, 0);
    rtype_t **ityps = alloca(rank * sizeof(rtype_t *));

    if(!vt)
        return NULL;
    for(int i=0; i<rank; i++)
    {
        ityps[i] = arg_type(args, i+1);
        if(!ityps[i])
            return NULL;
    }

    arr_chk_t chk = { .kinds = alloca(rank * sizeof(idx_kind)) };

    if(!check_arr_fetch(&chk, vt, ityps, rank))
        return NULL;
```

```
    if(phase == TRANSFER)
        return chk.typ;
    return (rank == 2 && chk.rank == 0) ? chk.typ : NULL;
}

static cresult trans_fetch(opt_phase phase, cnode_t *node, cell_t *cell,
                           const cbuiltin_t *bi, cnode_array_t *args)
{
    if(!check_arg_names(node) || alen(args) < 2)
        return SUCCESS;

    rtype_t *typ = (alen(args) == 2)
        ? trans_vec_fetch(phase, args)
        : trans_arr_fetch(phase, args);

    if(typ)
    {
        if(phase == TRANSFER)
            cell_set_type(cell, typ);
        else //(phase == TRANSFORM)
            call_become_builtin(node, bi, typ, typ);
        return CHANGED;
    }
    return SUCCESS;
}

static cresult enforce_fetch(cnode_t *node)
{
#ifndef NDEBUG
    rtype_t *type = node->builtin.optype;
    cnode_array_t *args = &node->builtin.args;

    assert(type);
    assert(cnode_compat(aref(args, 1), r_type_int));
    assert(alen(args) != 3 || cnode_compat(aref(args, 2), r_type_int));
    assert(cnode_compat(aref(args, 0), rvec_type_create(type)) ||
        cnode_compat(aref(args, 0), rarr_type_create(type)));
#endif
    return SUCCESS;
}

static void gen_fetch(cnode_t *node)
{
    bool is_mat = (alen(&node->builtin.args) == 3);
    // OP.type(.uni) dest, obj, idx[, idx]
    if(rtype_is_scalar(node->builtin.optype))
        asm_op_type(is_mat ? OP_getel2 : OP_getelt, node->builtin.optype);
    else
        asm_op(is_mat ? OP_getel2ptr : OP_geteltptr);
    asm_stack(loc_for(node));
    asm_args(&node->builtin.args);
}

static rtype_t *trans_vec_store(opt_phase phase, cnode_array_t *args)
{
    rtype_t *vt = arg_type(args, 0);
    rtype_t *rt = arg_type(args, 1);
    rtype_t *it = arg_type(args, 2); // not an error; see ‘[=‘ signature
```

```
    if(!vt || !rt || !it)
        return NULL;

    idx_chk_t chk = { 0 };

    if(!check_store(&chk, vt, it, rt))
    {
        //cnode_t *v = aref(args, 0);
        //...
        return NULL;
    }

    // '[=' passes through the replacement value, just like '='
    if(phase == TRANSFER)
        return rt;
    // optype is the container element type; we will convert arg1 if necessary
    return (chk.kind == SINGLE) ? vt->elt : NULL;
}

static rtype_t *trans_arr_store(opt_phase phase, cnode_array_t *args)
{
    int rank = alen(args) - 2;
    rtype_t *vt = arg_type(args, 0);
    rtype_t *rt = arg_type(args, 1);
    rtype_t **ityps = alloca(rank * sizeof(rtype_t *));

    if(!vt || !rt)
        return NULL;

    for(int i=0; i<rank; i++)
    {
        ityps[i] = arg_type(args, i+2);
        if(!ityps[i])
            return NULL;
    }

    arr_chk_t chk = { .kinds = alloca(rank * sizeof(idx_kind)) };

    if(!check_arr_store(&chk, vt, ityps, rank, rt))
        return NULL;
    if(phase == TRANSFER)
        return rt;
    return (rank == 2 && chk.rank == 0) ? vt->elt : NULL;
}

static cresult trans_store(opt_phase phase, cnode_t *node, cell_t *cell,
                           const cbuiltin_t *bi, cnode_array_t *args)
{
    if(!check_arg_names(node) || alen(args) < 3)
        return SUCCESS;

    rtype_t *typ = (alen(args) == 3)
        ? trans_vec_store(phase, args)
        : trans_arr_store(phase, args);

    if(typ)
    {
```

```
        if(phase == TRANSFER)
        {
            cell_set_type(cell, typ);
        }
        else // phase == TRANSFORM
        {
            // the opcode for this builtin does not need a new location
            // allocated, so replace us with the replacement value in
            // our users
            cnode_replace_in_users(node, aref(args, 1));
            // node.decl type not actually used
            call_become_builtin(node, bi, typ, typ);
        }
        return CHANGED;
    }
    return SUCCESS;
}

static cresult enforce_store(cnode_t *node)
{
    cnode_array_t *args = &node->builtin.args;

    assert(node->builtin.optype);
    assert(!cnode_is_used(node));
    assert(cnode_compat(aref(args, 2), r_type_int));
    assert(alen(args) != 4 || cnode_compat(aref(args, 3), r_type_int));
    // convert replacement value to optype
    return enforce_decl(node, node->builtin.optype, aptr(args,1), true);
}

static void gen_store(cnode_t *node)
{
    bool is_mat = (alen(&node->builtin.args) == 4);
    // OP.width obj, rpl, idx[, idx]
    asm_op_width(is_mat ? OP_setel2 : OP_setelt, node->builtin.optype);
    asm_args(&node->builtin.args);
}

const builtin_ops_t fetch_ops = { trans_fetch, enforce_fetch, gen_fetch,
                                  .is_pure = cons_is_pure };
const builtin_ops_t store_ops = { trans_store, enforce_store, gen_store,
                                  .is_void = true };

// container constructors
// FIXME just have the one, and determine what exactly to do based on
// bi->ops or whatever
static cresult trans_vector(opt_phase phase, cnode_t *node, cell_t *cell,
                            const cbuiltin_t *bi, cnode_array_t *args)
{
    if(!check_arg_names(node))
        return SUCCESS;
    assert(alen(args) > 0);

    cell_t *arg = cell_for(aref(args, 0));

    if(cell_const_obj(arg) && phase == TRANSFER)
    {
        robject_t *etyp = cell_const(arg);
```

```
        if(r_typeof(etyp) == r_type_type)
        {
            cell_set_type(cell, rvec_type_create((rtype_t *)etyp));
            // although doesn't actually matter, as we are not transforming
            return CHANGED;
        }
    }
    return SUCCESS;
}

static cresult trans_array(opt_phase phase, cnode_t *node, cell_t *cell,
                           const cbuiltin_t *bi, cnode_array_t *args)
{
    if(!check_arg_names(node))
        return SUCCESS;
    assert(alen(args) > 0);

    cell_t *arg = cell_for(aref(args, 0));

    if(cell_const_obj(arg) && phase == TRANSFER)
    {
        robject_t *etyp = cell_const(arg);

        if(r_typeof(etyp) == r_type_type)
        {
            cell_set_type(cell, rarr_type_create((rtype_t *)etyp));
            return CHANGED; // although doesn't really matter, as we are not transforming
        }
    }
    return SUCCESS;
}

const builtin_ops_t vector_ops = { trans_vector, NULL, NULL, .is_pure = cons_is_pure };
const builtin_ops_t array_ops = { trans_array, NULL, NULL, .is_pure = cons_is_pure };
```

Appendix C

Utilities

This appendix contains utility code used internally by the system.

Lists

⟨*list.h*⟩≡

```
/*
  untyped doubly-linked list abstraction

  _heavily_ inspired by include/linux/list.h,
  which is licensed under the

  GNU GENERAL PUBLIC LICENSE
     Version 2, June 1991
*/

typedef struct list
{
    struct list *next;
    struct list *prev;
} list_t;

#define LIST_INIT(_self) { &(_self), &(_self) }

static inline bool list_isempty(list_t *list)
{
    return list->next == list && list->prev == list;
}

static inline void list_init(list_t *list)
{
    list->next = list;
    list->prev = list;
}

static inline bool list_isfirst(list_t *head, list_t *list)
{
    return list->prev == head;
}
static inline bool list_islast(list_t *head, list_t *list)
{
    return list->next == head;
}
```

```
static inline void _list_set(list_t *prev, list_t *list, list_t *next)
{
    list->next = next;
    list->prev = prev;
    next->prev = list;
    prev->next = list;
}

static inline void list_add(list_t *prev, list_t *list)
{
    _list_set(prev, list, prev->next);
}

static inline void list_add_before(list_t *next, list_t *list)
{
    _list_set(next->prev, list, next);
}

static inline void list_remove(list_t *list)
{
    list->next->prev = list->prev;
    list->prev->next = list->next;
    list_init(list);
}

// splice constraint: from must be a head. to may be a link.
static inline void list_splice_before(list_t *to, list_t *from)
{
    if(list_isempty(from))
        return;
    from->prev->next = to;
    from->next->prev = to->prev;
    to->prev->next = from->next;
    to->prev = from->prev;
    list_init(from);
}

static inline void list_splice_after(list_t *to, list_t *from)
{
    if(list_isempty(from))
        return;
    from->prev->next = to->next;
    from->next->prev = to;
    to->next->prev = from->prev;
    to->next = from->next;
    list_init(from);
}

#define list_foreach(_head, _ptr)               \
    for(_ptr = (_head)->next;                   \
        _ptr != _head;                          \
        _ptr = _ptr->next)

#define list_foreach_from(_head, _ptr)          \
    for(;                                       \
        _ptr != _head;                          \
        _ptr = _ptr->next)
```

```
#define list_foreach_safe(_head, _ptr, _tmp)            \
    for(_ptr = (_head)->next, _tmp = _ptr->next;        \
        _ptr != _head;                                  \
        _ptr = _tmp, _tmp = _ptr->next)

#define list_foreach_reverse(_head, _ptr)       \
    for(_ptr = (_head)->prev;                   \
        _ptr != _head;                          \
        _ptr = _ptr->prev)

static inline size_t list_length(list_t *list)
{
    list_t *ptr;
    size_t count = 0;
    list_foreach(list, ptr)
        count++;
    return count;
}

/*
   slightly more convenient (albeit less standards-safe) interface
   inspired by mesa's gallium/auxiliary/util/u_double_list.h

   hic sunt demons nasal, if _obj is not initialised
*/
#define list_entry(_link, _obj, _mem)           \
    (void *)((char *)(_link)                    \
             - ((char *)&(_obj)->_mem           \
                - (char *)(_obj)))              \

#define list_foreach_entry(_head, _ptr, _mem)                           \
    for(_ptr = NULL, _ptr = list_entry((_head)->next, _ptr, _mem);      \
        &_ptr->_mem != (_head);                                         \
        _ptr = list_entry(_ptr->_mem.next, _ptr, _mem))

#define list_foreach_entry_reverse(_head, _ptr, _mem)                   \
    for(_ptr = NULL, _ptr = list_entry((_head)->prev, _ptr, _mem);      \
        &_ptr->_mem != (_head);                                         \
        _ptr = list_entry(_ptr->_mem.prev, _ptr, _mem))

#define list_foreach_entry_from(_head, _ptr, _mem)                      \
    for(; _ptr && &_ptr->_mem != (_head);                               \
        _ptr = list_entry(_ptr->_mem.next, _ptr, _mem))

#define list_foreach_entry_safe(_head, _ptr, _tmp, _mem)        \
    for(_ptr = NULL,                                            \
            _ptr = list_entry((_head)->next, _ptr, _mem),       \
            _tmp = list_entry(_ptr->_mem.next, _ptr, _mem);     \
        &_ptr->_mem != (_head);                                 \
        _ptr = _tmp,                                            \
            _tmp = list_entry(_ptr->_mem.next, _ptr, _mem))

#define list_while_entry(_head, _ptr, _mem)                             \
    for(_ptr = NULL, _ptr = list_entry((_head)->next, _ptr, _mem);      \
        &_ptr->_mem != (_head);                                         \
        _ptr = list_entry((_head)->next, _ptr, _mem))

/*
```

```
  simple singly-linked list abstraction
*/

#define SLIST(typ) typ *
#define slist_push(head, elt, link)             \
    ((elt)->link = (head), (head) = (elt))
#define slist_pop(head, link)                                   \
    ((head) ? _take_ptr((void **)&(head), (head)->link) : NULL)
#define slist_while_pop(head, ptr, link)        \
    while((ptr = slist_pop(head, link)))
#define slist_remove_head(head, link)           \
    ((head) = (head)->link)
#define slist_insert(before, after, link)                       \
    ((after)->link = (before)->link, (before)->link = (after))
#define slist_foreach(head, ptr, link)          \
    for(ptr = (head); ptr; ptr = ptr->link)
#define slist_foreach_safe(head, ptr, tmp, link)                \
    for(ptr = (head); ptr && (tmp = ptr->link, true); ptr = tmp)
#define slist_nreverse(head, link) do {                 \
        for(void *tmp = NULL, *ptr = (head); ptr; )     \
        {                                               \
            head = ptr;                                 \
            ptr = head->link;                           \
            head->link = tmp;                           \
            tmp = head;                                 \
        }                                               \
    } while(0)
```

Arrays

⟨*array.h*⟩≡

```
#define ARRAY(_typ)                             \
    struct                                      \
    {                                           \
        size_t length;                          \
        size_t size;                            \
        _typ *ptr;                              \
    }

#define ARRAY_INIT { 0, 0, NULL }

static inline void *
_adjust_array(void *ptr, size_t elt, size_t size)
{
    if(size == 0)
    {
        if(ptr)
            xfree(ptr);
        return NULL;
    }
    if(ptr)
        return xrealloc(ptr, elt * size);
    return xmalloc(elt * size);
}

static inline void *
_extend_array(size_t *plength, size_t *psize, unsigned n,
```

```
              void *ptr, size_t elt)
{
    *plength += n;
    if(*plength > *psize)
    {
        *psize = max(*psize, 1);
        while(*plength > *psize)
            *psize *= 2;
        return _adjust_array(ptr, elt, *psize);
    }
    return ptr;
}

static inline void *
_init_array(size_t *plength, size_t *psize, size_t elt, size_t init)
{
    *psize = init;
    *plength = 0;
    if(init != 0)
        return _adjust_array(NULL, elt, init);
    return NULL;
}

#define _size_elt(_arr) (sizeof(*((_arr)->ptr)))

#define array_init(_arr, _init)                                         \
    ((_arr)->ptr = _init_array(&(_arr)->length,                         \
                               &(_arr)->size,                           \
                               _size_elt(_arr),                         \
                               _init))

#define array_alloc(_arr, _init)                                \
    (*_arr = xmalloc(sizeof(*(*_arr))),                         \
     array_init(*_arr, _init))

#define array_fini(_arr) do                     \
    {                                           \
        assert(_arr);                           \
        if((_arr)->ptr)                         \
            free((_arr)->ptr);                  \
        (_arr)->ptr = NULL;                     \
    } while (0)

#define array_free(_arr) do                     \
    {                                           \
        array_fini(_arr);                       \
        free(_arr);                             \
    } while (0)

// resize the array exactly
#define array_shrink(_arr)                                  \
    ((_arr)->ptr = _adjust_array((_arr)->ptr,               \
                                 _size_elt(_arr),           \
                                 alen(_arr)))

// resize the array exactly, clear the array, return the contents pointer
#define array_cede(_arr)                        \
    (array_shrink(_arr),                        \
```

```
     array_clear(_arr),                         \
     _take_ptr((void **)&(_arr)->ptr, NULL))

#define array_clear(_arr) ((_arr)->length = 0)

#define array_isempty(_arr) ((_arr)->length == 0)

#define atail(_arr) ((_arr)->length - 1)

// XXX does not zero the additional storage
#define array_extend(_arr, _n)                                      \
    ((_arr)->ptr = _extend_array(&(_arr)->length,                   \
                                 &(_arr)->size, _n,                 \
                                 (_arr)->ptr,                       \
                                 _size_elt(_arr)))

#define array_push(_arr, _val)                                      \
    (array_extend(_arr, 1),                                         \
     (_arr)->ptr[atail(_arr)] = (_val))

#define array_pop(_arr)                                 \
    (assert(!array_isempty(_arr)),                      \
     (_arr)->ptr[--(_arr)->length])

#define array_drop(_arr, _n) ((_arr)->length -= _n)

#define array_take(_arr, _ptr)                                      \
    (!array_isempty(_arr) ?                                         \
     (*(_ptr) = (_arr)->ptr[--(_arr)->length],                      \
      true) :                                                       \
     false)

#define array_remove(_arr, _i) do                                   \
    {                                                               \
        assert(_i < (_arr)->length);                                \
        memmove((_arr)->ptr + (_i), (_arr)->ptr + (_i) + 1,         \
                ((_arr)->length - (_i) - 1) * _size_elt(_arr));     \
        (_arr)->length--;                                           \
    } while (0)

// insert val before i
#define array_insert(_arr, _i, _val) do                             \
    {                                                               \
        assert((_i) <= (_arr)->length);                             \
        array_extend(_arr, 1);                                      \
        if((_i) < atail(_arr))                                      \
           memmove((_arr)->ptr + (_i) + 1, (_arr)->ptr + (_i),      \
                   ((_arr)->length - (_i) - 1) * _size_elt(_arr));  \
        (_arr)->ptr[_i] = _val;                                     \
    } while (0)

// length = size = new
#define array_resize(_arr, _new) do                                 \
    {                                                               \
        (_arr)->ptr = _adjust_array((_arr)->ptr,                    \
                                    _size_elt(_arr),                \
                                    _new);                          \
        if((_new) > (_arr)->length)                                 \
```

```
            memset((_arr)->ptr + (_arr)->length, 0,                     \
                   ((_new) - (_arr)->length) * _size_elt(_arr));        \
        (_arr)->length = (_arr)->size = _new;                           \
    } while (0)

// element sizes must match
#define array_copy(_to, _from) do                                       \
    {                                                                   \
        assert(_size_elt(_from) == _size_elt(_to));                     \
        (_to)->length = (_from)->length;                                \
        (_to)->ptr = _adjust_array((_to)->ptr,                          \
                                   _size_elt(_to),                      \
                                   (_to)->size = (_from)->length);      \
        memcpy((_to)->ptr, (_from)->ptr,                                \
               (_from)->length * _size_elt(_from));                     \
    } while (0)

// FIXME unused?
#define array_concat(_to, _from) do                             \
    {                                                           \
        unsigned _total = (_to)->length + (_from)->length;      \
        assert(_size_elt(_from) == _size_elt(_to));             \
        if(_total > (_to)->size)                                \
            (_to)->ptr = _adjust_array((_to)->ptr,              \
                                       _size_elt(_to),          \
                                       (_to)->size = _total);   \
        memcpy((_to)->ptr + (_to)->length, (_from)->ptr,        \
               (_from)->length * _size_elt(_from));             \
        (_to)->length = _total;                                 \
    } while (0)


// trivial
// indexed only
#define array_foreach(_arr, _i)                 \
    for(_i = 0;                                 \
        _i < alen(_arr);                        \
        ++_i)

// pointer to element
#define array_foreach_ptr(_arr, _ptr)                   \
    for(_ptr = (_arr)->ptr;                             \
        _ptr < (_arr)->ptr + alen(_arr);                \
        ++_ptr)

// only set _entry in the test clause,
// otherwise it will happily index off the end of the array
#define array_foreach_entry(_arr, _entry)                       \
    for(int _i = 0;                                             \
        _i < alen(_arr) && (_entry = (_arr)->ptr[_i], true);    \
        ++_i)

#define array_map(_arr, _fn, _ptr) do           \
    {                                           \
        int _i;                                 \
        for(_i=0; _i<(_arr)->length; _i++)      \
            _fn((_arr)->ptr[_i], _i, _ptr);     \
    } while(0)
```

```
#define alen(_arr) ((_arr)->length)
#define asize(_arr) ((_arr)->size)
#define adata(_arr) ((_arr)->ptr)

#ifdef NDEBUG
#define aref(_arr, _i) ((_arr)->ptr[_i])
#define aset(_arr, _i, _val) ((_arr)->ptr[_i] = (_val))
#define aptr(_arr, _i) ((_arr)->ptr + _i)
#else
// FIXME: double evaluation makes these unsafe for side-effecting arguments
#define aref(_arr, _i) (assert(_i < (_arr)->length), (_arr)->ptr[_i])
#define aset(_arr, _i, _val) (assert(_i < (_arr)->length), (_arr)->ptr[_i] = (_val))
#define aptr(_arr, _i) (assert(_i < (_arr)->length), (_arr)->ptr + _i)
#endif
```

Worklist

⟨*worklist.h*⟩≡

```
// fixed-length ring buffer with presence bitmap
// XXX stretchy buffer is a relatively simple extension (hah)
// note that, since the maximum index is specified, and no item can be
// present more than once, overflow can't occur
#define WORKLIST(typ) struct {                          \
        int len, count, idx;                            \
        bitmap_t *map;                                  \
        typ *ptr;                                       \
    }

static inline int _wrap(int v, int l)
    { return (v >= l) ? v-l : v; }

#define worklist_isempty(wl) ((wl)->count == 0)

#define worklist_init(wl, n) do {                       \
        assert(n > 0);                                  \
        (wl)->len = n;                                  \
        (wl)->count = 0;                                \
        (wl)->idx = 0;                                  \
        (wl)->map = bitmap_create(n);                   \
        (wl)->ptr = xcalloc(n, sizeof(*((wl)->ptr)));   \
    } while(0)

#define worklist_fini(wl) do {                          \
        xfree((wl)->ptr);                               \
        xfree((wl)->map);                               \
    } while(0)

#define worklist_push(wl,v,id) do {                     \
        if(!bitmap_get_bit((wl)->map, id))              \
        {                                               \
            (wl)->ptr[_wrap((wl)->idx + (wl)->count,    \
                            (wl)->len)] = v;            \
            (wl)->count++;                              \
            bitmap_set_bit((wl)->map, id);              \
        }                                               \
    } while(0)
```

```
#define worklist_take(wl,p,id)                          \
    (!worklist_isempty(wl) ?                            \
     (*p = (wl)->ptr[(wl)->idx],                        \
      (wl)->count--,                                    \
      (wl)->idx = _wrap((wl)->idx + 1, (wl)->len),      \
      bitmap_clear_bit((wl)->map, id),                  \
      true) :                                           \
     false)
```

General

⟨*util.c*⟩≡

```
#include "global.h"
#include <sys/mman.h>

void _vfatal(const char *fn, const char *fmt, va_list va)
{
    fprintf(stderr, "%s: ", fn);
    vfprintf(stderr, fmt, va);
    fprintf(stderr, "\n");
    fflush(stderr);
    abort();
}

void _fatal(const char *fn, const char *fmt, ...)
{
    va_list va;
    va_start(va, fmt);
    _vfatal(fn, fmt, va);
    va_end(va); /* NOTREACHED */
}

void *xrealloc(void *ptr, size_t size)
{
    if(size == 0)
        fatal("allocation size 0.");
    void *ret = realloc(ptr, size);
    if(!ret)
        fatal("allocation failed.");
    return ret;
}

// FIXME: attribute malloc for these
void *xmalloc(size_t size)
{
    if(size == 0)
        fatal("allocation size 0.");
    void *ret = malloc(size);
    if(!ret)
        fatal("allocation failed.");
    return ret;
}

void *xcalloc(size_t num, size_t size)
{
    if(size == 0)
```

```
        fatal("allocation size 0.");
    void *ret = calloc(num, size);
    if(!ret)
        fatal("allocation failed.");
    return ret;
}

void xfree(void *ptr)
{
    if(!ptr)
        fatal("freeing NULL.");
    free(ptr);
}

void *xmap(size_t sz)
{
    void *ret = mmap(NULL, sz, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
    if(ret == MAP_FAILED)
    {
        perror("mmap failed");
        exit(1);
    }
    return ret;
}

bool string_equal(const void *x, const void *y)
{
    return !strcmp((char *)x, (char *)y);
}
uint32_t string_hash(const void *ptr)
{
    const char *string = ptr;
    return hash_code(string, strlen(string));
}
bool ptr_eq(const void *x, const void *y)
{
    return x == y;
}
uint32_t ptr_hash(const void *ptr)
{
    return hash_code(&ptr, sizeof(ptr));
}

void hexdump(FILE *fp, uint8_t *mem, ptrdiff_t len, unsigned width)
{
    do
    {
        for(int i=0; (!width || i<width) && len>0; i++, len--)
            fprintf(fp, "%02X ", *mem++);
        if(width)
            fprintf(fp, "\n");
    } while (len > 0);
}

void mem_dump(uint8_t *mem, ptrdiff_t len)
{
    printf("%p: ", mem);
    hexdump(stdout, mem, len, sizeof(intptr_t));
```

```
}

// static return buffer, caveat caller
char *mem_string(uint8_t *mem, ptrdiff_t len)
{
    static char buf[32];
    char *ptr = buf;

    while(len-- > 0)
    {
        ptr += snprintf(ptr, sizeof(buf) - (ptr-buf), "%02X ", *mem++);
    }

    return buf;
}
```

⟨*util.h*⟩≡

```
/*
  double expansion - stringification or token-pasting
  disables argument expansion in function-like macros.
  so, wrap the operation in another macro to get the
  desired (expanded and then stringified/pasted) result.
*/
#define STRINGIFY(a) #a
#define PASTE2(a, b) a##b
#define PASTE3(a, b, c) a##b##c
#define PASTE4(a, b, c, d) a##b##c##d
#define PASTE5(a, b, c, d, e) a##b##c##d##e
#define PASTE6(a, b, c, d, e, f) a##b##c##d##e##f
/*
  (triple expansion - for token-pasting parameterised by an
  object-like macro)
*/
#define PPASTE2(a, b) PASTE2(a, b)
#define PPASTE3(a, b, c) PASTE3(a, b, c)

// caveat: double evaluation!
#ifndef max
#define max(a,b) ((a) > (b) ? (a) : (b))
#endif
#ifndef min
#define min(a,b) ((a) < (b) ? (a) : (b))
#endif

#define lengthof(a) (sizeof(a) / sizeof(*a))

// reasonably standards-safe; thanks stddef.h
#define container_of(ptr, typ, field)           \
    ((typ *)((char *)(ptr) - offsetof(typ, field)))

// caveat: assumes all pointers are the same size!
static inline void *_take_ptr(void **ptr, void *next)
{
    void *tmp = *ptr;
    *ptr = next;
    return tmp;
}

// define NOSPAM per-file: dbg output suppressed for that file.
```

```
// define DBG_SUPPRESS in config.mk: no dbg output.
// define NDEBUG: no asserts, no dbg output
#if !defined(NDEBUG) && !defined(NOSPAM) && !defined(DBG_SUPPRESS)
#define DBG(args...) fprintf(stderr, args)
#define DBGPRINT(val) r_print(stderr, val)
#define DBG_OUTPUT
#else
#define DBG(args...) do { } while(0)
#define DBGPRINT(val) do { } while(0)
#endif

// NOTE: gcc-ism
#define fatal(fmt, args...) _fatal(__func__, fmt , ##args)
#define vfatal(fmt, va) _vfatal(__func__, fmt, va)
void _vfatal(const char *fn, const char *fmt, va_list va);
void _fatal(const char *fn, const char *fmt, ...);

void *xrealloc(void *ptr, size_t size);
void *xmalloc(size_t size);
void *xcalloc(size_t num, size_t size);
void xfree(void *ptr);
void *xmap(size_t sz);

// bit vector
typedef uint8_t bitmap_t;
static inline size_t size_for_bits(unsigned n)
    { return (n+7)>>3; }
static inline bool bitmap_get_bit(bitmap_t *bits, unsigned n)
    { return (bits[n>>3] & (1<<(n&7))) != 0; }
static inline uint8_t bitmap_get_byte(bitmap_t *bits, unsigned n)
    { return bits[n>>3]; }
static inline void bitmap_set_bit(bitmap_t *bits, unsigned n)
    { bits[n>>3] |= 1<<(n&7); }
static inline void bitmap_clear_bit(bitmap_t *bits, unsigned n)
    { bits[n>>3] &= ~(1<<(n&7)); }
static inline void bitmap_free(bitmap_t *bits)
    { xfree(bits); }
static inline void bitmap_reset(bitmap_t *bits, unsigned n)
    { memset(bits, 0, size_for_bits(n)); }
static inline bitmap_t *bitmap_create(unsigned n)
    { return xcalloc(1, size_for_bits(n)); }

// XXX int __builtin_popcount(unsigned int)
static inline unsigned bitmap_count_bits(bitmap_t *bits, unsigned n)
{
    unsigned r = 0;
    for(int i=0; i<size_for_bits(n); i++)
    {
        uint8_t byte = bits[i];
        for(int j=0; j<8; j++, byte >>= 1)
        {
            if(byte & 1)
                r++;
        }
    }
    return r;
}
```

```
void hexdump(FILE *fp, uint8_t *mem, ptrdiff_t len, unsigned width);
void mem_dump(uint8_t *mem, ptrdiff_t len);
char *mem_string(uint8_t *mem, ptrdiff_t len);

bool string_equal(const void *x, const void *y);
uint32_t string_hash(const void *ptr);
bool ptr_eq(const void *x, const void *y);
uint32_t ptr_hash(const void *ptr);
```

$\langle$*global.h*$\rangle\equiv$

```
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include <alloca.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <float.h>
#include <math.h>
#include <setjmp.h>
#include <unistd.h>
// debuggery
#include <mcheck.h>
#include <assert.h>

#include "util.h"
#include "hash.h"
#include "array.h"
#include "list.h"
#include "rt/runtime.h"
#include "compiler.h"
```

Appendix D

Hash Tables

This appendix contains an open-addressing linear-probing hash table which is used, by the system, as a map and a set. The `hash_code_seed` function is from Appleby (2008).

Interface

⟨*hash.c*⟩≡

```
#include "global.h"

static inline uint32_t rotl32(uint32_t x, int8_t r)
{
    return (x << r) | (x >> (32 - r));
}

static uint32_t fmix(uint32_t h)
{
    h ^= h >> 16;
    h *= 0x85ebca6b;
    h ^= h >> 13;
    h *= 0xc2b2ae35;
    h ^= h >> 16;

    return h;
}

uint32_t hash_code(const void *key, size_t len)
{
    return hash_code_seed(key, len, 0);
}

uint32_t hash_code_seed(const void *key, size_t len, uint32_t h1)
{
    const uint8_t *data = (const uint8_t *)key;
    const int nblocks = len / 4;

    const uint32_t c1 = 0xcc9e2d51;
    const uint32_t c2 = 0x1b873593;

//----------
// body

    const uint32_t *blocks = (const uint32_t *)(data + nblocks*4);
```

```
    for(int i = -nblocks; i; i++)
    {
        uint32_t k1 = blocks[i];

        k1 *= c1;
        k1 = rotl32(k1,15);
        k1 *= c2;

        h1 ^= k1;
        h1 = rotl32(h1,13);
        h1 = h1*5+0xe6546b64;
    }

//----------
// tail

    const uint8_t *tail = (const uint8_t *)(data + nblocks*4);

    uint32_t k1 = 0;

    switch(len & 3)
    {
    case 3: k1 ^= tail[2] << 16;
    case 2: k1 ^= tail[1] << 8;
    case 1: k1 ^= tail[0];
        k1 *= c1; k1 = rotl32(k1,15); k1 *= c2; h1 ^= k1;
    };

//----------
// finalization

    h1 ^= len;

    return fmix(h1);
}

typedef struct hashbase
{
    size_t nentries;
    size_t size;
    uint32_t (*hash_fn)(const void *key);
    bool (*eq_fn)(const void *x, const void *y);
} hashbase_t;

static void hash_init(hashbase_t *hash,
                      uint32_t (*hash_fn)(const void *),
                      bool (*eq_fn)(const void *, const void *))
{
    *hash = (hashbase_t) {
        .size = 4,
        .nentries = 0,
        .hash_fn = hash_fn,
        .eq_fn = eq_fn
    };
}

// map of void * -> void *
#define HASH_NAME hashmap
```

```
#define VALUE_TYPE void *
#include "hash.c.inc"
#undef HASH_NAME
#undef VALUE_TYPE
// set of void *
#define HASH_NAME hashset
#include "hash.c.inc"
#undef HASH_NAME
```

⟨*hash.h*⟩≡

```
uint32_t hash_code(const void *key, size_t len);
uint32_t hash_code_seed(const void *key, size_t len, uint32_t h1);

// map of void * -> void *
#define HASH_NAME hashmap
#define VALUE_TYPE void *
#include "hash.h.inc"
#undef HASH_NAME
#undef VALUE_TYPE

// set of void *
#define HASH_NAME hashset
#include "hash.h.inc"
#undef HASH_NAME
```

Implementation

⟨*hash.c.inc*⟩≡

```
// type and function names
#define hash_t PPASTE2(HASH_NAME, _t)
#define node_t PPASTE2(HASH_NAME, _node_t)
#define func(n) PPASTE3(HASH_NAME, _, n)

#if defined(VALUE_TYPE)
// map
#define result_t VALUE_TYPE
#define result_found(n) (n)->value
#define args(k, v) k, v
typedef struct
{
    void *key;
    VALUE_TYPE value;
} node_t;
//
#else // defined(VALUE_TYPE)
// set
#define result_t void *
#define result_found(n) n->key
#define args(k, v) k
typedef struct
{
    void *key;
} node_t;
//
#endif // defined(VALUE_TYPE)

#define no_result ((result_t)0)
```

```
typedef struct HASH_NAME
{
    hashbase_t h;
    node_t *entries;
} hash_t;

static void func(rehash)(args(const void *key, void *value), void *ptr)
{
    func(insert)((hash_t *)ptr, args(key, value));
}

static bool func(resize)(hash_t *tbl, size_t newsize)
{
    hash_t tmp = *tbl;

    assert(newsize >= tbl->h.nentries);

    tmp.h.nentries = 0;
    tmp.h.size = newsize;
    tmp.entries = xcalloc(newsize, sizeof(node_t));

    func(map)(tbl, func(rehash), (void *)&tmp);
    assert(tmp.h.nentries == tbl->h.nentries);

    tbl->h.size = newsize;
    xfree(tbl->entries);
    tbl->entries = tmp.entries;

    return true;
}

// returns the value of nonnull key, or NULL if not present
// if found is nonnull, sets it to true or false on return.
// (only makes sense if VALUE_TYPE is set)
result_t func(get)(const hash_t *tbl, args(const void *key, bool *found))
{
    unsigned mask, index;
    node_t *node;

    if(!key)
        goto fail;

    mask = tbl->h.size - 1;
    index = tbl->h.hash_fn(key) & mask;

    while(node = &tbl->entries[index], node->key)
    {
        if(tbl->h.eq_fn(node->key, key))
        {
#if defined(VALUE_TYPE)
            if(found)
                *found = true;
#endif
            return result_found(node);
        }
        index = (index + 1) & mask;
    }
```

```
fail:
#if defined(VALUE_TYPE)
    if(found)
        *found = false;
#endif
    return no_result;
}


result_t func(add_update)(hash_t *tbl, args(const void *key, const void *value),
                          bool add, bool update)
{
    uint32_t hash;
    unsigned mask, index;
    node_t *node;

    if(!key)
        return no_result;

    if(add && ((float)tbl->h.nentries / tbl->h.size > 0.7))
    {
        if(!func(resize)(tbl, tbl->h.size<<1))
            fatal("can't resize hash table.\n");
    }

    mask = tbl->h.size - 1;
    hash = tbl->h.hash_fn(key);
    index = hash & mask;

    while(true)
    {
        node = &tbl->entries[index];
        if(!node->key)
        {
            if(add)
            {
                node->key = (void *)key;
#if defined(VALUE_TYPE)
                node->value = (void *)value;
#endif
                tbl->h.nentries++;
                return result_found(node);
            }
            return no_result;
        }
        else if(tbl->h.eq_fn(node->key, key))
        {
            if(update)
            {
                result_t old = result_found(node);
#if defined(VALUE_TYPE)
                node->value = (void *)value;
#endif
                return old;
            }
            //return no_result;
            return result_found(node);
        }
```

```
        index = (index + 1) & mask;
    }
    return no_result; /* NOTREACHED */
}

// remove nonnull key and its value from tbl, if present
// inspired substantially by http://en.wikipedia.org/wiki/Open_addressing
result_t func(remove)(hash_t *tbl, const void *key)
{
    unsigned mask, index, gap, natural;
    node_t *node;
    result_t result;

    if(!key)
        return no_result;

    if(tbl->h.size > 4 && (float)tbl->h.nentries / tbl->h.size < 0.3)
    {
        if(!func(resize)(tbl, tbl->h.size>>1))
            fatal("can't resize hash table.\n");
    }

    mask = tbl->h.size - 1;
    index = tbl->h.hash_fn(key) & mask;
    result = no_result;

    while(node = &tbl->entries[index], node->key)
    {
        if(tbl->h.eq_fn(node->key, key))
        {
            gap = index;
            result = result_found(node);
        }
        else if(result)
        {
            // the natural location of this node, which is actually at index
            natural = tbl->h.hash_fn(node->key) & mask;

            // if the gap lies in this node's chain, swap them.
            // index only advances, so gap is always before it in sequence order
            // this is just one test - (natural .before. gap)
            if((gap < index) ?
               (natural <= gap || natural > index) :
               (natural <= gap && natural > index))
            {
                tbl->entries[gap] = *node;
                gap = index;
            }
        }
        index = (index + 1) & mask;
    }

    if(result)
    {
        node = &tbl->entries[gap];
        node->key = NULL;
#if defined(VALUE_TYPE)
```

```
        node->value = no_result;
#endif
        tbl->h.nentries--;
        return result;
    }

    return no_result;
}

hash_t *func(create)(uint32_t (*hash_fn)(const void *),
                     bool (*eq_fn)(const void *, const void *))
{
    hash_t *tbl = xmalloc(sizeof(hash_t));

    hash_init(&tbl->h, hash_fn, eq_fn);
    tbl->entries = xcalloc(tbl->h.size, sizeof(node_t));

    return tbl;
}

void func(free)(hash_t *hash)
{
    xfree(hash->entries);
    xfree(hash);
}

void func(clear)(hash_t *tbl, size_t newsize)
{
    tbl->h.nentries = 0;
    if(newsize > 0)
    {
        tbl->h.size = newsize;
        xfree(tbl->entries);
        tbl->entries = xcalloc(newsize, sizeof(node_t));
    }
    else
        memset(tbl->entries, 0, tbl->h.size * sizeof(node_t));
}

// not stable under insertion or deletion!
void func(map)(const hash_t *tbl,
               void (*fn)(args(const void *, void *), void *),
               void *ptr)
{
    int i;
    node_t *node;

    for(i=0,node=tbl->entries; i<tbl->h.size; i++,node++)
    {
        if(node->key)
            fn(args(node->key, node->value), ptr);
    }
}

#undef hash_t
#undef node_t
#undef func
#undef result_t
```

```
#undef result_found
#undef args
```

⟨*hash.h.inc*⟩≡

```
// shared with hash_c.h
#define hash_t PPASTE2(HASH_NAME, _t)
#define func(n) PPASTE3(HASH_NAME, _, n)
#if defined(VALUE_TYPE)
#define result_t VALUE_TYPE
#define args(k, v) k, v
#else
#define result_t void *
#define args(k, v) k
#endif

typedef struct HASH_NAME hash_t;

result_t func(get)(const hash_t *tbl, args(const void *key, bool *found));
result_t func(add_update)(hash_t *tbl, args(const void *key, const void *value),
                          bool add, bool update);
result_t func(remove)(hash_t *tbl, const void *key);
void func(map)(const hash_t *tbl,
               void (*fn)(args(const void *, void *), void *),
               void *ptr);
hash_t *func(create)(uint32_t (*hash_fn)(const void *),
                     bool (*eq_fn)(const void *, const void *));
void func(free)(hash_t *tbl);
void func(clear)(hash_t *tbl, size_t newsize);

// returns old value if already present
static inline result_t
func(insert)(hash_t *tbl, args(const void *key, const void *value))
{
    return func(add_update)(tbl, args(key, value), true, false);
}

// only update value if present, returning old value. else, just return zero.
static inline result_t
func(update)(hash_t *tbl, args(const void *key, const void *value))
{
    return func(add_update)(tbl, args(key, value), false, true);
}

// add or update. returns old value if present, new value if not
static inline result_t
func(set)(hash_t *tbl, args(const void *key, const void *value))
{
    return func(add_update)(tbl, args(key, value), true, true);
}

#undef hash_t
#undef result_t
#undef func
#undef args
```

Appendix E

Bibliography

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: An optimizing compiler for scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 219–233. ACM, 1986. doi: 10.1145/12276.13333.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

Austin Appleby. Murmurhash3, 2008. URL `https://github.com/aappleby/smhasher/wiki/MurmurHash3`. [Accessed at 14/01/2018].

Richard A. Becker and John M. Chambers. *S: an Interactive Environment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole, 1984. ISBN 053403313X.

Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language: A Programming Environment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole, 1988. ISBN 0-534-09192-X.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, March-April 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2010.118.

Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. URL `http://arxiv.org/abs/1209.5145`.

Ake Björck. Solving linear least squares problems by Gram-Schmidt orthogonalization. *B.I.T*, 7:1–21, 1967.

Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoît Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. Technical report, INRIA, 2008.

Carl Friedrich Bolz, Antonio Cuni, Maciej Fija, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, Genova, Italy, 2009. ACM. ISBN 978-1-60558-541-3. doi: 10.1145/1565824.1565827. URL `http://portal.acm.org/citation.cfm?id=1565827`.

Florian Brandner, Benoit Boissinot, Alain Darte, Benoît Dupont de Dinechin, and Fabrice Rastello. Computing liveness sets for SSA-form programs. Technical Report 7503, INRIA, 2011.

Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice & Experience*, 27(6):701–724, June 1997. doi: 10.1002/(SICI)1097-024X(199706)27:6⟨701::AID-SPE104⟩3.3.CO;2-S.

Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice & Experience*, 28(8):859–881, July 1998. doi: 10.1002/(SICI)1097-024X(19980710)28:8⟨859::AID-SPE188⟩3.0.CO;2-8.

Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 246–257. ACM, 1995. doi: 10.1145/207110.207154.

William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 128–139. ACM, 1994. doi: 10.1145/182409.156786.

Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. Technical Report TR-06-33870, Department of Computer Science, Rice University, 2011.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):415–490, October 1991. doi: 10.1145/115372.115320.

Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978. doi: 10.1145/359642.359655.

M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *J. Instruction-Level Parallelism*, 5, 2003.

GNU General Public License, June 1991. URL `http://www.gnu.org/licenses/gpl.html`.

GNU M4, December 2017. URL `https://www.gnu.org/software/m4`.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005. doi: 10.3217/jucs-011-07-1159.

Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. doi: 10.1080/10618600.1996.10474713.

Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 2012. ISBN 978-1-4200-8279-1.

Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, March 1995. doi: 10.1145/202530.202532.

Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. ISBN 0-201-89683-4.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO'04, Palo Alto, California, Mar 2004.

Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1): 121–141, July 1979. doi: 10.1145/357062.357071.

Robert A. MacLachlan. The python compiler for CMU Common Lisp. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 235–246. ACM, 1992. doi: 10.1145/141471.141558.

Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998. doi: 10.1145/272991.272995.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, April 1960. doi: 10.1145/367177.367199.

Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

Hanspeter Mössenböck and Michael Pfeiffer. Linear scan register allocation in the context of ssa form and register constraints. In R. Nigel Horspool, editor, *Compiler Construction*, pages 229–246. Springer Berlin Heidelberg, 2002. doi: 10.1007/3-540-45937-5_17.

Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100. ACM, 2007. doi: 10.1145/1250734.1250746.

R Core Team. The R project for statistical computing, June 2016. URL `https://www.r-project.org`. version 3.3.1.

R Core Team. The R project for statistical computing, November 2017. URL `https://www.r-project.org`. version 3.4.3.

Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Static Analysis: Second International Symposium, SAS '95 Glasgow, UK, September 25–27, 1995 Proceedings*, pages 366–381. Springer Berlin Heidelberg, 1995. doi: 10.1007/3-540-60360-3_50.

Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. Technical Report CMU-CS-90-127, Carnegie Mellon University, September 1990.

Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000. ISSN 1388-3690. doi: 10.1023/A:1010000313106.

Luke Tierney. Compiling R: A preliminary report. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, DSC 2001, March 2001.

Luke Tierney. A byte code compiler for R, November 2016. URL `http://homepage.divms.uiowa.edu/~luke/R/compiler/compiler.pdf`. [Accessed at 31/01/2018].

Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, SIGPLAN '98, pages 142–151. ACM, June 1998.

P. M. Vaidya. An optimal algorithm for the all-nearest-neighbors problem. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 117–122, Oct 1986. doi: 10.1109/SFCS.1986.8.

S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. doi: 10.1109/MCSE.2011.37.

Guido van Rossum. Python reference manual. Technical Report CS-R9525, Centrum voor Wiskunde en Informatica, Amsterdam, May 1995.

Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991. doi: 10.1145/103135.103136.

Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 170–179. ACM, 2010. doi: 10.1145/1772954.1772979.

Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 132–141. ACM, 2005. doi: 10.1145/1064979.1064998.