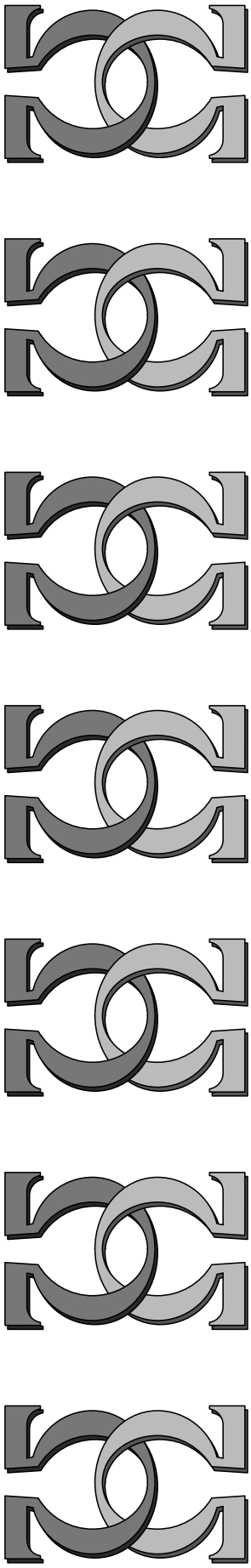

CDMTCS
Research
Report
Series

Can We Solve the Pipeline Problem?

C. S. Calude[1], **A. Coull**[2],
J. P. Lewis[2,3]

[1]University of Auckland, NZ
[2]Weta Digital, Wellington, NZ
[3]Victoria University, Wellington, NZ

CDMTCS-471
October 2014

Centre for Discrete Mathematics and
Theoretical Computer Science

Can We Solve the Pipeline Problem?

Cristian S. Calude[1], Alasdair Coull[2], J. P. Lewis[2,3*†]
[1]Department of Computer Science, University of Auckland, New Zealand
[2]Weta Digital, Wellington, New Zealand
[3]School of Engineering and Computer Science, Victoria University, Wellington, New Zealand

Abstract

A list of the most important unsolved problems in VFX production should include the word "pipeline". Pipelines are a major source of efficiency, but they require ongoing development, and are seemingly never finished. They also resist change. Is it possible to automatically generate pipelines given desired inputs and outputs or design one "universal" pipeline that can do everything with no further modification ever needed? We present similarities between pipelines and programs, and recall metamathematical statements which show that the answer to the above question is *negative*. However, new approximate solutions may mitigate to a large extent this negative situation.

1 Introduction

In computer graphics production, "pipeline" refers to the collection of scripts and processes that streamline passing data between programs and across departments. It often includes version control and dependency tracking capabilities. "Pipeline" is sometimes also considered to include the artists who embody a facility's knowledge of these processes. Pipelines are frequently developed on-demand and incrementally, and are often proprietary, although some commercial pipeline solutions do exist.

A pipeline is a major source of efficiency. It automates regular aspects of workflow, reducing errors. As one example of the potential for such error, consider the work of a lighting technical director. S/he works during the day adjusting lights and shader settings while using either a preview renderer or low-resolution settings suitable for rapid exploration. Before leaving for the day, the artist switches the render settings to appropriate high-resolution values and launches an overnight render on the farm. The problem is that there may be 10 or more such parameters that have to be changed (image resolution, shading rate and other sample rates, file paths, etc.). A pipeline component can encapsulate these parameter changes as a single "interactive" versus "hires" switch and thereby reduce the potential that some changes are overlooked.

Pipelines are a benefit, but also a problem. A mature pipeline grows to (literally) millions of lines of code, representing tens of person-years of work. Pipelines also resist innovation: the introduction of new software requires not just completing and debugging software, but also adapting the pipeline to incorporate it. One might say that a poorly designed pipeline makes a company efficient at doing what it did in the past.

In this note we consider the (somewhat obvious) observation that pipelines share many characteristics with programs. This suggests that terms and concepts from programming can be applied to pipelines—as is already done in practice. It also suggests, however, that developing a pipeline is intrinsically difficult. Metamathematical arguments show that programming cannot be fully automated, hence pipeline development cannot be fully automated too. However, recent results—which mitigate this negative statement—open the door to tools allowing significant possibilities of automation.

2 Pipelines are (a lot) like programs

A number of components of pipelines resemble or make use of programming constructs, for example:

Iteration. Visual effects involves iterative development. The assets that comprise each shot are iteratively improved and shown to a visual effects supervisor until approval.

Conditional Branching. Data can conditionally pass through different parts of the pipeline. For example, consider a character whose clothing is simulated. If the clothing interpenetrates, the simulation may be rerun with adjusted inputs. However if the simulation takes hours to run and the shot is due tomorrow, the shot may instead be sent to the paint department to fix.

Polymorphism. Level of detail is a type of polymorphism. A single model may exist in several different resolutions suitable for either different view distances, or different purposes (animation, simulation, rendering), and the pipeline software may automatically retrieve the correct version for the purpose.

Structures. A character asset may consist of the geometry at various resolutions, together with associated textures, shaders, and other things. These objects travel together through the pipeline, resembling a C++ structure.

Variables. Objects may be retrieved by name from a database.

Specific Algorithms. Pipeline design involves algorithmic choices. For example, the render queue, and the workflow as a whole, may be processed as a priority queue (the general job scheduling problem is known to be NP-hard [Garey and Johnson 1979]). If N software packages are in use (Lightwave, Softimage, Houdini, etc.), it will save development time to have a star topology with a common interchange format and up to N converters, rather than implementing the up to $N(N-1)/2$ converters required to directly convert every package's file format into every other.

Parallelism. A good pipeline is designed to allow parallel execution.

The reader can probably think of additional examples of parallels and similarities between pipelines and programs.

Conditional branching and variables are generally enough to establish Turing completeness, so we see that the computational power of pipelines is equivalent to that of Turing machines. This equivalence is not superficial or accidental. The various programming constructs listed above were introduced into pipelines in order to effectively solve the problems at hand.

*Authors are alphabetically listed. Lewis is the lead author.

†Presented at *Digipro2014*, `http://olm.co.jp/digipro2014/program`.

3 Undecidable problems in mathematics and programming

Given the analogy between pipelines and programs, we will make a further comparison, to mathematics. It is possible to setup correspondences between mathematics and computation, for example [Svozil 1993]

axioms $\Longleftrightarrow$	program input or initial state
rules of inference $\Longleftrightarrow$	program interpreter
theorem(s) $\Longleftrightarrow$	program output
derivation $\Longleftrightarrow$	computation

Note that there are several ways of defining the correspondence between formal and computational systems. For example, the Curry-Howard isomorphism involves "natural deduction style" rather than "Hilbert style" proofs. Also the division of a computational system into "computer" and "program" can be done in different ways. Regardless, the correspondences can be made rigorous and allow metamathematical statements from one domain to be translated to the other.

Metamathematical *incompleteness* statements describe the ultimate limits of what can be proved or computed. Curiously, a number of the incompleteness theorems resemble classic paradoxes. For example, the Berry paradox involves a phrase of the form "the smallest integer that cannot be defined in less than 20 words", which has used 12 words to identify the integer in question. This paradox is related to the Chaitin incompleteness theorem in algorithmic information theory [Calude 2002], that says that if a formal system can be described in n bits, it cannot prove statements that assert that a particular string (i.e. bitpattern) has algorithmic complexity much larger than n. Incompleteness in computer science takes the form of undecidable problems—questions that do not have an algorithmic solution. The most (in)famous is the halting problem: *there is no program which can decide in a finite time whether an arbitrarily given program (as input) eventually stops or not.*

The halting problem is occasionally regarded as an isolated mathematical curiosity, but, in fact, it is the reason many programming problems (like debugging) have no algorithmic solutions. In fact most questions about program and programming are undecidable. For example Rice's theorem states that there is no algorithmic way of deciding any non-trivial externally observable property of programs. Under straightforward assumptions it is not possible to objectively estimate the time needed to develop programs [Lewis 2001]. The maximum run time of a program that halts rises faster than any computable function of the size of that program [Cooper 2004]. Undecidable problems not only exist, but there are "everywhere" [Calude et al. 1994].

For our purposes the Hilbert decision problem—does there exist an algorithm that can prove or disprove every mathematical statement?—is the central undecidable problem. The negative answer—which is a consequence of the undecidability of the halting problem—was proved by Church and Turing [Cooper 2004]. It shows that mathematics cannot be automated, hence, because of the mathematics-computation analogy, programming cannot be fully automated (a result sometimes called sometimes the "full employment theorem").

4 Does it matter?

New pipeline development is driven by both desired efficiency gains and by the need to support tools that solve new problems. The tools available are getting better and better. If at some hypothetical point in the future there are no new visual effects problems, pipelines will nevertheless remain in development—until we reach the point where every movie can be made automatically or in a short time by a single person.

If the equivalence between pipelines and programs, described above, is granted, the full employment theorem asserts that pipeline problem is undecidable: there is no general method to automatically generate pipelines (given desired inputs and outputs) nor a single "universal" pipeline that can do everything. This is an *abstract negative* result. Does it matter for the practice of pipeline development?

The boundary between solvable and undecidable problems is being mapped with recent results: *Most programs which do not halt quickly, never stop* [Calude and Stay 2008]. This fact was used to design anytime algorithms—programs that exchange execution time for quality of results [Grass 1996]—which provide approximate (with arbitrary precision) solutions to the halting problem [Calude and Desfontaines 2014]. Proof-assistants, like Coq or Isabelle, are software tools for the development of formal proofs by human-machine collaboration [Wikipedia 2014]. These positive results extend the boundary of what is known to be possible and mitigate the impact of undecidability.

At present our discussion remains general and abstract. Practical program design debates consider specific questions such as whether delegation is preferable to inheritance, whether C++ templates are worth the trouble, or whether Python is better than Perl.[1] While discussing these complexities, it is perhaps easy to imagine that our problems will be "solved" if the next software system incorporates better decisions with regard to these polarizing choices. The merit of examining our problem in full generality is that it abstracts from these particulars and remind us that *programming is intrinsically hard* (while also indicating what will be needed to make profound progress). This essay has shown that developing pipelines is as hard as programming and as such it is a problem worthy of significant resources and expert minds.

References

CALUDE, C. S., AND DESFONTAINES, D., 2014. Anytime algorithms for non-ending computations. U. Auckland *CDMTCS Research Report* 463.

CALUDE, C. S., AND STAY, M. A. 2008. Most programs stop quickly or never halt. Advances in Applied Mathematics 40, 295–308.

CALUDE, C., JÜRGENSEN, H., AND ZIMAND, M. 1994. Is independence an exception? *Applied Mathematics and Computation 66*, 1, 63–76.

CALUDE, C. S. 2002. *Information and Randomness: An Algorithmic Perspective*, 2nd ed. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

COOPER, S. 2004. *Computability Theory*. Chapman Hall/CRC, London.

GAREY, M. R., AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

GRASS, J. 1996. Reasoning about computational resource allocation: An introduction to anytime algorithms. *Crossroads 3*, 1 (Sept.), 16–20.

LEWIS, J. P. 2001. Large limits to software estimation. *ACM Software Engineering Notes 26*, 4, 54–59.

[1] It is.

SVOZIL, K. 1993. *Randomness & Undecidability in Physics*. World Scientific, Singapore.

WIKIPEDIA, 2014. Proof assistant — Wikipedia, the free encyclopedia. Online: accessed 21 June.