Black Hole Computation

Matias Scharager

Carnegie Mellon University

mscharag@cs.cmu.edu

1 Introduction

Consider the universe we live in right now. We are all governed by several scientific laws of nature such as Newton's laws and many more. These laws create harmony in our universe, creating a natural system of order. We can say these laws restrict possible states of being as if the universe is being statically type-checked. We can measure states of being as we progress forwards in time in our universe, but there are also many things far beyond our comprehension. One of these things is potential parallel universes which can be hidden inside theoretical wormholes in black holes.[10]

All of this is simply a metaphor for programming languages. We start by defining the laws that govern the universe: the structural operational semantics. We can also prove the harmony and safety of these laws to show that our universe is stable in existence. Once this is achieved, we can write code in our language that type-checks, essentially the matter of our universe, and watch how it behaves as it executes.

In fact, every time we create a new programming language, we are defining a parallel universe![9] The way of reasoning in one language is different from the methodology of another language. This extends far beyond a simple syntax renaming: the information that can exist in the universe is altered due to the fundamental rules of the language.

In this paper, we take a look at the wormholes between programming language universes in an attempt to understand the interaction between these universes via logic and data transfer. In doing so, we uncover ingenious ways to perform computation, utilizing the computational power of alternate universes. This lens also allows understanding old ideas in a new way, suggesting alternative ways of creating compilers.

2 Compilers

Surprisingly, there is already a commonly used name for wormholes between universes: compilers. Compilers are structures that convert the logic and computational content in one language to that of another. As such, they act as a one-way ticket from a source to the target language.

Typically, compilers are considered separate entities that are not part of the programming language universe itself. The fact that we need to cross so many universe boundaries makes the compilation extremely inefficient in maintaining the expressive capacity of the program.

2.1 SML $\rightarrow$ C $\rightarrow$ TAL

Let's compile a Standard ML (SML)[4] program into Typed Assembly Language (TAL).[5] We write a compiler in C to do this conversion. If we think of each of these programming languages to be their own separate universe, then the C compiler exists in the C universe while the source code exists in the SML universe. The goal is to utilize the object of the C universe to teleport the SML code into the TAL universe.

We are capable of seeing the SML universe object in the C universe. This is done in a very clumsy way: through a syntax protocol. The SML code is expressed on paper via defining syntax arbitrarily for the different components of the language. This is then saved to a text file which is a rudimentary logic that

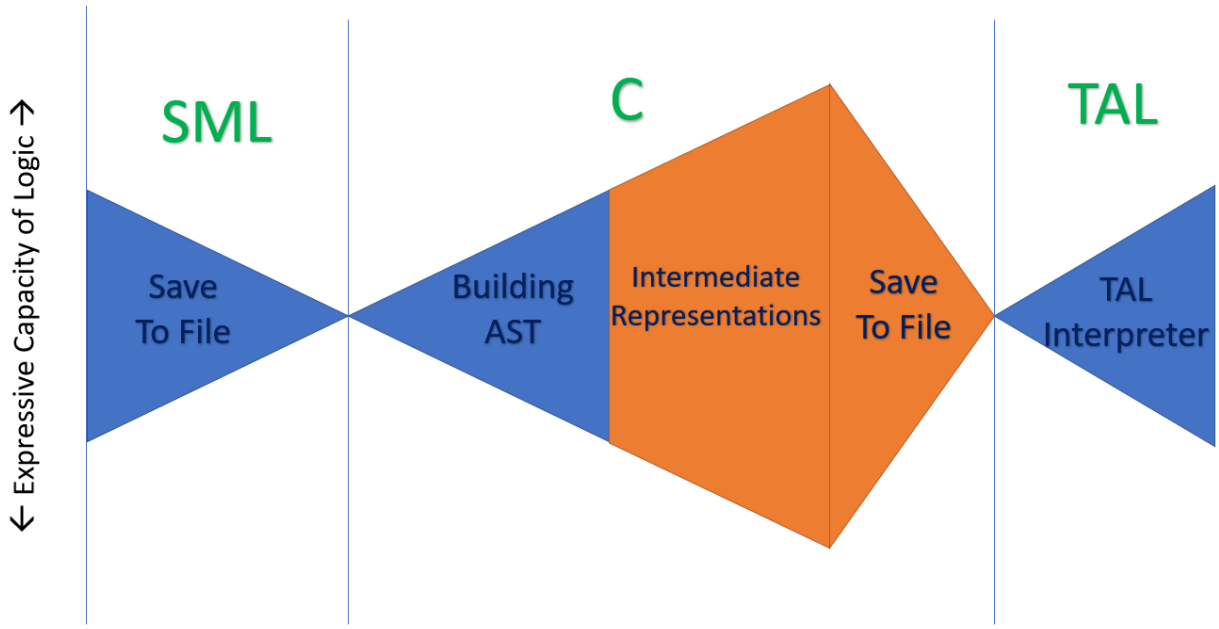

Figure 1: Compilation Expressiveness

exists in all of the programming languages universes. Then the C compiler takes this file and operates on it via a parser. This is known as creating an abstract syntax tree (AST).

Consider the logical interpretation present at each stage of this transportation. We start in the SML universe with all the expressive power of SML, then we strip away all the expressiveness by converting it into a text file. This is a thinning down of the logic to make it easy to transfer. Upon entering the C universe, the compiler attempts to give back meaning to the text it is parsing, creating a simulated version of the SML universe to gain back some of the logic we had before. This thinning down and building back up of logic seems redundant.

We are not done here yet. We then create various intermediate representations and logics of the program we are compiling, defining various logical principles in the C universe. TAL expresses things in a different way than SML does, so we need to simulate all the changes in logic via multiple phases of compilation, increasing the logical interpretation of the program itself. Once all this is done, we save it to another file in the arbitrary syntax defined for TAL.

This out-file is transferred to a separate universe again, the TAL universe, where it can be executed in a comprehensible way on real-world computers. There can be further manipulations of this code as we need to increase our logical interpretation from the out-file to execution, but the details of this are beyond the scope of this paper.

The overall process can be seen in Figure 1. Notice that the expressiveness of the code at the beginning and at the end is roughly equivalent: we want our code to run exactly what we were planning on running right at the start. In between, there are a lot of inefficiencies where we lose expressiveness only to make an effort to gain it back up. Also, note the tightness of the informational content we can send through the barrier between universes. This is due to the only shared component between all these languages being a joint file system.

2.2 Removing First Boundary

The most direct attempt to salvage expressiveness across the program boundary is to implement the SML compiler in SML itself. If we naively attempt this, the result is that we still need to save the program to a file, destroying the expressive ability, only to convert it back into an AST to manipulate. Even if we simply

converted the program into a "string" object instead of a file, we would still be losing informational content.

Instead, we must think in a more direct way. We want to convert our SML program into an SML AST of the same expressive content. The easiest conversion is the identity function: the SML program *is* an SML AST. Thankfully, the SML datatype system is a very nice way to write SML code in SML, and, moreover, the SMLNJ standard library includes an AST library! The library documentation can be found here: https://www.smlnj.org/doc/Compiler/pages/ast.html.

All SML programs that are normally written in SML syntax can alternatively be expressed in this AST framework with minimal impact on the code development experience. Moreover, we can easily convert already existing SML programs into this AST framework, allowing for forward compatibility.

To demonstrate the easiness of utilizing this AST library, we interviewed all the SML code developers in software engineering roles in the US. There were 0 complaints out of the large sample size of 0 SML code developers, providing empirical evidence for how easy it is for these engineers to convert to this representation.

If we consider the same "black hole" metaphor we utilize throughout the paper, our new compiler now exists in the same universe as the original source code, saving us the cost of sending the information through the black hole. We can think of this as the black hole sucking up the whole source code universe.

One of the major benefits of this conversion is blurring the distinction between programming and compilation. Every program must be self-compiling, meaning it must describe the appropriate operations needed to compile the program into a lower-level language.

Note that while we opted to discuss removing the C component of compilation, it is just as easy to base everything in C by using an SML AST library written in C to express our language. However, this would require coding in C, specifically writing an SML compiler in C, so it was instantly discarded as a bad idea.

2.2.1 Alternative Strategies

Since the current suggested framework for self compilation is rather complicated to implement, we can consider an extension to the SML language that includes an in-line compiler as a language feature. For example, here is the proposed hello world program.

```
COMPILE(
    print "Hello World!"
)
```

Notice that this COMPILE monad is admissible into our language, as we are capable of providing a computational interpretation of compilation. As such, we recover valid structural operational semantics without having to modify the type safety proofs of the language. As such, we are capable of expressing compilation as a program effect: it prints the compiled program to the output file.

There are several modifications we can make to this framework that would allow for greater user experiences. For one, we can allow for multiple kinds of COMPILE monads, representing which compiler flags we wish to have in different parts of the program. This will allow for a very hands-on approach to compiler optimizations, as we can mark which areas we want to have compiled with what strategies. We can even allow for a mix of the default COMPILE monads along with user implemented compilation so that we have the greatest amount of flexibility.

Once again, we are drawn back to our black hole analogy where monads represent universe teleportation. We are expressly marking which chunks of our current universe (SML) we want to send to an alternative universe (TAL).

2.2.2 Quines

It is worth mentioning the relationship of this format of programming to that of quines. A quine is any program that prints itself as its output [1]. As such, we can think of quines as a degenerate case of compilation where we do not perform any compilation steps, and simply spit back the original source code. Note that there is an additional concept of "multiquines" where compilation does in fact occur as one language is changed into another language [3], but ultimately, multiquines go back to the original source language after a finite number of steps, ruining the usefulness of the conversion.

2.3 Removing the Second Boundary

While we have now established a wonderful programming methodology to eliminate the first mistake of compilation, we have yet to explain how we wish to run the self-compiling program.

One option is to have a meta compiler, which compiles the self-compiling program. However, as Thompson suggests in Reflections on Trusting Trust[7], this is a very risky move, as the meta compiler could have been compiled by a meta meta compiler that has a trojan horse in it, essentially breaking the authenticity of our self compiling program. To make matters worse, if we choose to implement a meta compiler ourselves as a self compiling program, we would still need to write a meta meta compiler ourselves to compile the meta compiler, and we would end up writing an infinite hierarchy of compilers before we can safely run our program. This is essentially Russell's paradox in the form of compilation. It would be fair to say that there exists some magical black box compiler that exists as our initial self-compiling compiler, but such a thing would have to implement in an "engineering hack" kind of way in practice.

However, we are still in luck. The programming language we are expressing our program in has a valid small step relation, describing the process of executing. As such, we can conjecture the existence of a machine that would perform this small step relation as a morphism of our program, slowly progressing until we have reached a final stopping state at which we have achieved assembly output.

Unfortunately, such a perfect computer cannot exist in the real world with modern-day computer architecture, as it would require an infinite number of state transitions to model all possible whole program expressions a program can have during execution. As such, I would like you to take a brief moment of silence while reading this paper to lament the fact that our idealized computer is purely an ideal. Thank you for your moment of silence.

Let us take a step back and find out what exactly a computer in modern-day computer architecture is capable of computing. It is reasonable to claim that a computer can "run assembly code" meaning it is capable of performing operations that model the operations that assembly makes. The validity of this claim is left as an exercise to the reader. This means that we automatically have an assembly interpreter given to us magically.[8]

To remove the second boundary, we must implement an SML AST representation in an assembly program, then write an SML compiler in the assembly language to compile the SML ASTs that we write. Since everything is already in assembly, and we have a valid way of running assembly programs on a computer, then we have successfully preserved expressive capacity throughout the whole compilation process.

2.3.1 Syntactic Sugar

It is worth noting the definition of "syntactic sugar"[2] to assure ourselves that this compilation strategy for SML into assembly is not just syntactic sugar for assembly. Syntactic sugar is merely an admissible rule of computation being added into the program without defining a new structural operational semantics. In our setup, even though we are expressing SML ASTs in assembly, we are still distinguishing the structural operational semantics of SML from those of assembly, giving us a different set of expressiveness tools that can't be easily represented in assembly.

Note that under this definition of syntactic sugar, languages that do not have well-defined structural operational semantics can be considered to be just glorified assembly, since we claim that assembly is the only language we are capable of interpreting on a computer in practice. Aside from structural operational semantics, it seems sufficient to define a language via a specification such as C and thus a C to assembly compiler is a compiler, however the analytical tools we have for program analysis and compiler correctness would have to be vastly different in such a setting so it is beyond the scope of this paper.

3 Parallel Computation

If this were a sensible research paper, the term parallel computation would mean multi-threaded cores with fork and join operations, or even some parallelized dynamics formalization of the structural operational semantics. In our case, parallel computation means none of these things, yet at the same time means all of these things! Parallel is derived from the "parallel universes" that run concurrently to our current universe of existence. Parallel computation, therefore, means utilizing parallel universes for computation.

While our current universe is constrained to a certain set of logical principles, these principles do not necessarily apply to the parallel universe. As such, sending information to the parallel universe, and running the computations in that universe instead, allows us to save quite a lot of computation time among other things. We will now cover various useful parallel universes and their abilities.

3.1 Effect-Free Effects

It is often the case that we want to reason about the state of our environment. This might involve various operations including IO file reading, direct input from the user, reading and writing to memory, or utilizing cache or registers. However, these are always pesky to reason about from a programming language theory perspective.

We exclude non-termination in our definitions (we will cover this later separately), and define a pure function as one that does not reason about the state of the environment in any way. The advantage to this is the mathematical and category-theoretic idea that the function always returns the same thing, and is simply a mapping of input values to output values.

This means an impure function, otherwise known as an effectful function, is one that performs any of these IO operations. Clearly by the name "impure" we can see that this is an inferior version of pure functions. The advantage to it is we can do these "dirty" effectful computations, and this can save us computation time. For example, we all know from software engineering internship interviews that if we aren't sure how to make our solution faster, we yell out the words "hash set" or "dynamic programming" and hope that one of these two things is what the interviewer wants to hear.

We can go through the effort of making some of these operations pure but this usually requires a painstaking amount of monads and weird things that frequent the nightmares of software developers. Instead, we can simply create a universe where all these effects can exist, and so to calculate effectful functions, we send the data and run the program in this parallel universe. Since none of these effects occur in our current universe, we are never accessing the state of our environment, so such a function is considered pure.

With this, we recover all the niceties of pure functions within the current universe's structural operational semantics, and we handle the effects through a trivial lemma: since there are infinitely many parallel universes, there is one that has a nice behaving environment with reads and writes that we can send data to.

3.2 Infinity, Non-Termination, and Undecideability

There are several problems in computer science that are very important that we wish to compute but haven't found the means to. For example, some very important research questions include "what is the biggest natural number," "what is the last digit of pi," and "does this arbitrary Turing machine terminate on this input."

It is trivial to write a function that computes the largest natural number utilizing continuation-passing style:

```
datatype n = Z | S of n
open SMLofNJ.Cont

fun biggest () =
    let
        fun big (f : n -> n) = big (fn x => f (S x))
    in
        callcc (fn k => big (throw k))
    end

val ans = biggest()
```

Unfortunately, executing this program results in non-termination as this function infinitely loops. We would ideally like to keep running this program until it terminates, but since our universe is constrained to running things in a finite amount of time, we will never find the answer we want to find.

However, parallel universes are not constrained in the same way as our universe. We can find a parallel universe that runs an infinite number of steps of the program instantaneously and then send this program to be run in that universe. In such a universe, the concept of the last natural number can be made a reality, along with the last digit of pi, so we will refer to this universe as Infinity Universe.

We all know that the last question we asked is the famous Halting Problem. Fundamentally, this problem is undecideable, which means that no algorithm can provide a yes or no solution to it. How then can we correctly state that an algorithm in Infinity Universe gives us the answer to an undecideable problem? For this, we need a simple lemma: Infinity Universe allows for paradoxes. If time can be run infinitely instantaneously, then if we consider the current time that we are in as our starting point, then this is no different of a starting point than one second into the future, or a year, or a century, or even a larger time gap like the length of time it takes mankind to eradicate covid. This means that current time + 1 unit of time = current time. If we standardize our current time to being time zero, this equation shows us that $1 = 0$. Previous research [6] demonstrates a computer verified proof of arbitrary program termination in polynomial time in an unsound type system, solving the P=NP problem. In fact, we can prove any program halts, as we can simply run it in Infinite Universe, demonstrating that Infintite Universe is simply an unsound universe.

A quick note, I would rather not wish to exist in an unsound universe. I am uncertain as to what that would do to the basic laws of physics, but it is worth conducting experiments to determine this.

3.3 Implementing Parallel Universes

As we just discussed the extreme usefulness of these parallel universes, we would like a way of modeling them in our programming language. Again, we consider the black hole principle. We are currently in universe A which we know and comprehend (have structural operational semantics). We wish to run a program in universe B, where we know nothing about how things run. So we open up a black hole, and send the program into the black hole!

This black hole is a monad that allows for communication from universe A to universe B. We can define some construct DATA (or abbreviated E) that allows us to express a syntactic formulation of contents in universe B expressible in universe A. We can also define bind and return operations to express the monadic structure. Here is an example of how data might look like if universe B was a universe filled with cats.

$$M := x \mid () \mid \langle M, M \rangle \mid M \cdot 1 \mid M \cdot 2 \mid \lambda x : A.M \mid M\ M \mid \texttt{blackhole}(E)$$
$$E := \texttt{ret}(M) \mid \texttt{bnd}(M; x.E) \mid \texttt{MEOW} \mid \texttt{PURR} \mid \texttt{NYA} \mid E\ \texttt{owo}\ E$$

We can now express the program that explains the command to solve the Halting Problem in universe B.

$$\texttt{blackhole}(\texttt{NYA owo MEOW}) : \bigcirc(\texttt{HALTING PROBLEM})$$

As we can see, this is a valid term of the type $\bigcirc(\texttt{HALTING PROBLEM})$. It provides to us a function that solves the Halting Problem expressed in universe B.

We now witness a dilemma. How can this possibly be true? Well, universe B is not constrained to any meaningful principles that universe A obeys. However, the backlash to this is that no information in universe B is comprehensible in universe A. We can continue to transmit information to universe B from universe A via this black hole and continue to compute things like solving the Halting Problem in universe B, but the black hole is a one-way street. Once you've gone to a universe of cats, you will never want to leave.

However, this allows for quite some grand amount of interesting properties in the universe A. Since all of these $\texttt{blackhole}(E)$ terms reduce within the universe B instead of universe A, and none of the information is observable within universe A, then it is accurate to state that for all E, E', $\texttt{blackhole}(E) \simeq \texttt{blackhole}(E')$ where $\simeq$ represents Kleene equivalence over any observation, or can be expanded to be contextual equivalence. As such, we can recover the structural operational semantics of the term language M in universe A via the following reduction

$$\frac{}{\texttt{blackhole}(E) \mapsto \texttt{blackhole}(\bullet)} \qquad \frac{}{\texttt{blackhole}(\bullet)\ \texttt{val}} \qquad \frac{}{\Gamma \vdash \texttt{blackhole}(\bullet) : \bigcirc(A)}$$

Where $\bullet$ represents the fact that E was successfully transmitted to universe B. As such, progress and preservation are simple inductions on the static and dynamic rules, and we assure type safety.

3.3.1 Parallel Parallel Universes

There is no reason to limit ourselves to simply two universes as we have in our previous example. We can have infinitely many parallel universes, and continue to expand our language constructs via adding these extra parallel universes.

In fact, we every time we add an extra monad into our system, we are only making moderate corrections to the proof of type safety, along with all the logical relations associated with the programming language. This allows for an easy method of extending our language.

3.4 Recovering the Other-Worldly Data

Even though data sent through a black hole to a separate universe can never come back, it is still the case that something might come back to our universe if there is a black hole in the other universe connecting it to ours. As such, this suggests that a bidirectional message passing system between universes is possible.

It is interesting to draw ties from this idea to that of the π-calculus. For the most part, we consider the case where processes in the π-calculus correspond to the same universe so we can consider communication via channels in this language as black holes which simply return you to the same universe you started from. This restriction doesn't seem to be necessary, as different processes can utilize different operational semantics without breaking the abstractions of the π-calculus.

Constructing a system for message passing between multiple universes is beyond the scope of this paper, because it's already hard enough to implement the structural operational semantics for one language, let alone two or more. If we intend to pass messages between universes, we can no longer just hypothetically suppose the operations of one language existing, but must actually provide it in order to run the program.

4 Conclusions

As we discovered in this paper, we can implement a compilation strategy that preserves the expressiveness of the program logic throughout the whole compilation process. This is done via expressing every program we write as a self-compiling, self-interpreting program in the assembly language.

We also discovered a methodology for performing any arbitrary effectful or undecidable computation in constant time by transmitting the information content to a separate universe via a black hole monad. This paves the way for several interesting implementations in parallel universes that are observable via our own.

References

[1] Douglas R. Hofstadter. *Godel, escher, bach*. Basic Books, 1979.

[2] Peter J Landin. The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320, 1964.

[3] David Madore. Quines (self-replicating programs). `http://www.madore.org/~david/computers/quine.html`.

[4] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[5] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.

[6] Matias Scharager. Verified proof of p=np in the silence theorem prover language. In *SIGBOVIK 2020*, SIGBOVIK, pages 51–54, Pittsburgh, PA, 2020. Association for Computational Heresy.

[7] Ken Thompson. Reflections on trusting trust. In *ACM Turing award lectures*, page 1983. 2007.

[8] Wikipedia. Incantation. `https://en.wikipedia.org/wiki/Incantation`.

[9] Wikipedia. Isekai. `https://en.wikipedia.org/wiki/Isekai`.

[10] Wikipedia. Wormhole. `https://en.wikipedia.org/wiki/Wormhole`.