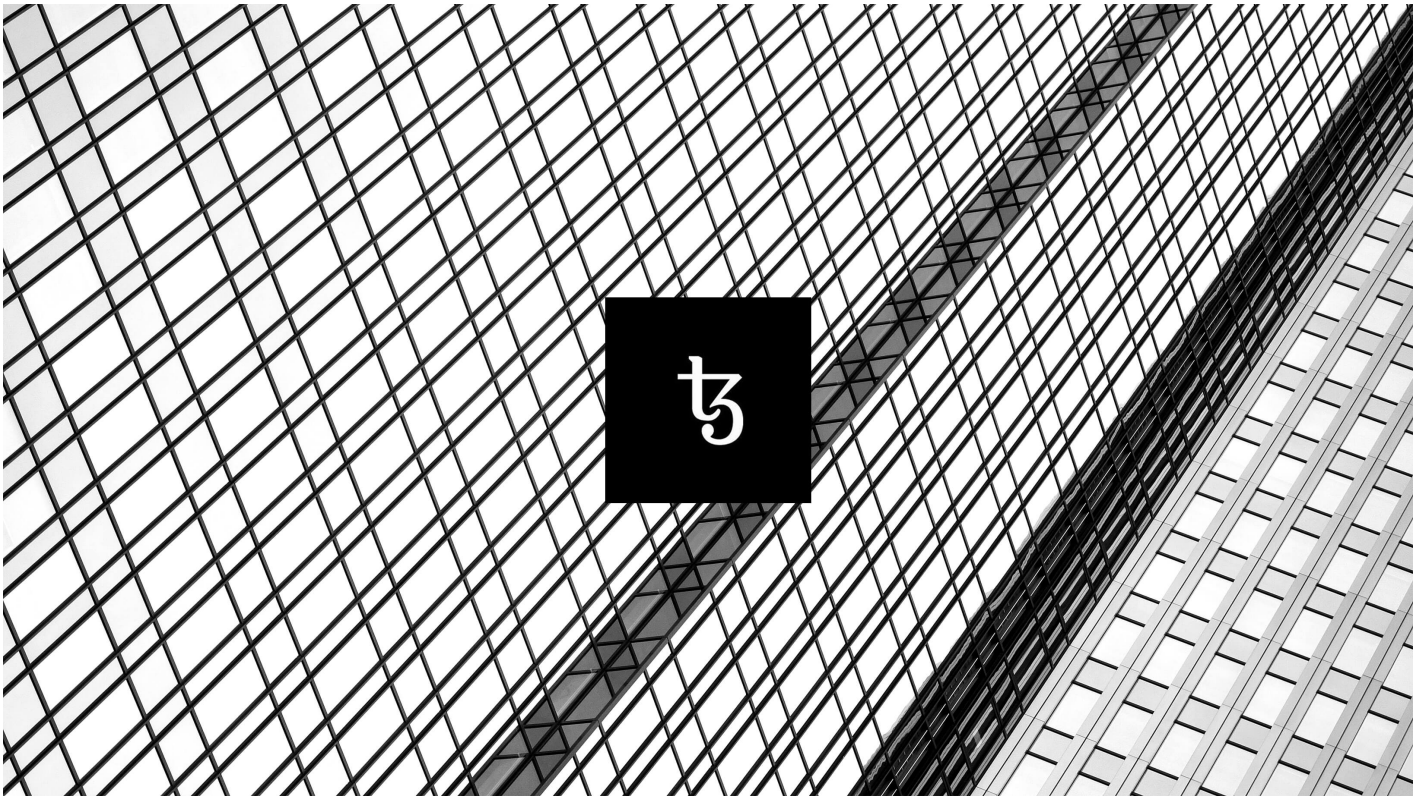
Simplicity and Michelson

A programming language that is too simple

DECEMBER 08, 2017 PHILIP WADLER 9 MIN READ



Simplicity

Only once in my life have I encountered a programming language that was too simple to use. That was Lispkit Lisp, developed by Peter Henderson, Geraint Jones, and Simon Jones, which I saw while serving as a postdoc at Oxford, 1983–87, and which despite its simplicity was used to implement an entire operating system. It is an indightment of the field of programming languages that I have not since encountered another system that I consider too simple. Until today. I can now add a second system to the list of those that are too simple, the appropriately-titled Simplicity, developed by Russell O'Connor of Blockstream. It is described by a paper here and a website here.

The core of Simplicity consists of just nine combinators: three for products (`pair`, `take`, and `drop`), three for sums (`injl`, `injr`, and `case`), one for unit (`unit`), and two for plumbing (`iden` and `comp`). It is throughly grounded in ideas from the functional programming, programming language, and formal methods communities.

When I call Simplicity too simple it is intended as a compliment. It is delightful to see full adders and cryptographic hash functions cobbled together using just products, sums, and units. It is eye-opening to see how far one can get without recursion or iteration, and how this enables simple analyses of the time and space required to execute a program. It is a confirmation to see a system with foundations in category theory and

sequent calculus. Now I know what to say when developers respond to my talk “Categories for the Working Hacker” by asking “But how can we use this in practice?”

The system is accompanied by a proof of its correctness in Coq, which sets a high bar for competing systems. O’Connor even claims to have a proof in Coq that the Simplicity implementation of SHA-256 matches the reference specification provided by Andrew Appel’s Verified Software Toolchain project (VST), which VST proved corresponds to the OpenSSL implementation of SHA-256 in C.

At IOHK, I have been involved in the design of Plutus Core, our own smart contract scripting language, working with Darryl McAdams, Duncan Coutts, Simon Thompson, Pablo Lamela Seijas, and Grigore Rosu and his semantics team. We have a formal specification which we are preparing for release. O’Connor’s work on Simplicity has caused us to rethink our own work: what can we do to make it simpler? Thank you, Russell!

That said, Simplicity is still too simple, and despite its emphasis on rigour there are some gaps in its description.

Jets

A 256-bit full adder is expressed with 27,348 combinators, meaning addition in Simplicity requires several orders of magnitude more work than the four 64-bit addition instructions one would normally use. Simplicity proposes a solution: any commonly used sequence of instructions may be abbreviated as a “jet”, and implemented in any equivalent matter. Hence, the 27,348 combinators for the 256-bit full adder can be ignored, and replaced by the equivalent four 64-bit additions.

All well and good, but this is where it gets too simple. No one can afford to be inefficient by several orders of magnitude. Hence, any programmer will need to know what jets exist and to exploit them whenever possible. In this sense, Simplicity is misleadingly simple. It would be clearer and cleaner to define each jet as an opcode. Each opcode could still be specified by its equivalent in the other combinators of Simplicity, but programs would be more compact, faster to execute, and—most important—easier to read, understand, and analyse accurately. If one ignores jets, the analyses of time and space required to execute a program, given toward the end of the paper, will be useless—off by orders of magnitude. The list of defined jets is given nowhere in the paper. Nor could I spot additional information on Simplicity linked to from its web page or findable by a web search. More needs to be done before Simplicity can be used in practice.

Gaps

It’s not just the definition of jets which is absent from the paper, and cannot be found elsewhere on the web. Lots more remains to be supplied.

- Sections 2.4, 2.5, 3.2 claim proofs in Coq, but apart from defining the semantics of the nine combinators in Appendix A, no Coq code is available for scrutiny.
- Section 2.5 claims a representation of Simplicity terms as a dag, but it is not specified. Lacking this, there is no standard way to exchange code written in Simplicity.
- Section 4.4 defines an extended semantics for Simplicity that can read the signature of the current transaction, support Merklised abstract syntax trees, and fail when a transaction does not validate. It also lifts meanings of core (unextended) Simplicity programs to the extended semantics. However, it says nothing about how the seven combinators that combine smaller Simplicity programs into bigger ones act in the extended semantics! It’s not hard to guess the intended definitions, but worrying that they were omitted from a paper that aims for rigour.
- Section 3 provides a Bit Machine to model the space and time required to execute Simplicity. The model is of limited use, since it ignores the several orders of magnitude improvement offered by jets. Further, the Bit Machine has ten instructions, enumerated on pages 10–12, but the list omits the vital “case” instruction which appears in Figure 2. Again, it’s not hard to guess, but worrying it was omitted.

Michelson

A second language for scripting blockchains is Michelson. It is described by a paper here and a website here. (Oddly, the website fails to link to the paper.)

I will offer just one word on Michelson. The word is: “Why?” Michelson takes many ideas from the functional programming community, including higher-order functions, data structures such as lists and maps, and static type safety.

Currently, it is also much more thoroughly described and documented than Simplicity. All of this is to be commended.

But Michelson is an inexplicably low-level language, requiring the programmer to explicitly manipulate a stack. Perhaps this was done so that there is an obvious machine model, but Simplicity offers a far superior solution: a high-level model for programming, which compiles to a low-level model (the Bit Machine) to explicate time and space costs.

Or perhaps Michelson is low-level to improve efficiency. Most of the cost of evaluating a smart contract is in cryptographic primitives. The rest is cheap, whether compiled or interpreted. Saving a few pennies of electricity by adopting an error prone language—where there is a risk of losing millions of dollars in an exploit—is a false economy indeed. Premature optimisation is the root of all evil.

The language looks a bit like all the bad parts of Forth and Lisp, without the unity that makes each of those languages a classic. Lisp idioms such as `CAAR` and `CDADAR` are retained, with new ones like `DUUP`, `DIIIIP`, and `PAAIAIAAIR` thrown in.

There is a fair set of built-in datatypes, including strings, signed and unsigned integers, unit, product, sum, options, lists, sets, maps, and higher-order functions. But there is no way for users to define their own data types. There is no way to name a variable or a routine; everything must be accessed by navigating a data structure on the stack.

Some operations are specified formally, but others are left informal. For lists, we are given formal rewriting rules for the first three operators (CONS, NIL, IF_CONS) but not the last two (MAP, REDUCE). Type rules are given in detail, but the process of type inference is not described, leaving me with some questions about which programs are well typed and which are not. It reminds me of a standard problem one sees in early work by students—the easy parts are thoroughly described, but the hard parts are glossed over.

If I have understood correctly, the inference rules assign types that are monomorphic, meaning each term has exactly one type. This omits one of the most successful ideas in functional programming, polymorphic routines that act on many types. It means back to the bad old days of Pascal, where one has to write one routine to sort a list of integers and a different routine to sort a list of strings.

Several of these shortcomings are also shared by Simplicity. But whereas Simplicity is intended as a compilation target, not to be read by humans, the Michelson documentation includes a large collection of examples suggesting it is intended for humans to write and read.

Here is one of the simpler examples from the paper.

```
{ DUP ; CDAAR ; # T
NOW ;
COMPARE ; LE ;
IF { DUP ; CDADR ; # N
     BALANCE ;
     COMPARE ; LE ;
     IF { CDR ; UNIT ; PAIR }
        { DUP ; CDDDR ; # B
          BALANCE ; UNIT ;
          DIIIP { CDR } ;
          TRANSFER_TOKENS ;
          PAIR } }
   { DUP ; CDDAR ; # A
     BALANCE ;
     UNIT ;
     DIIIP { CDR } ;
     TRANSFER_TOKENS ;
     PAIR } }
```

The comment `# T` is inserted as a reminder that `CDAAR` extracts variable `T`, and similarly for the other variables `N`, `B`, and `A`. This isn't the 1950s. Why don't we write `T` when we mean `T`, instead of `CDAAR`? WHY ARE WE WRITING IN ALL CAPS?

In short, Michelson is a bizarre mix of some of the best and worst of computing.

Conclusion

It is exciting to see ideas from the functional programming, programming languages, and formal methods communities gaining traction among cryptocurrencies and blockchains. While there are shortcomings, it is fantastic to see an appreciation of how these techniques can be applied to increase reliability—something which the multi-million dollar exploits against Ethereum show is badly needed. I look forward to participating in the conversations that ensue!

Postscript

The conversation has begun! Tezos have put up a page to explain Why Michelson. I've also learned there is a higher-level language intended to compile into Michelson, called Liquidity.