Axi Design Musings

Wojciech Kołowski

17 December 2024

Intro

The purpose of this presentation is to discuss Axi design in an informal setting. I want to bring some important considerations to the participants' collective consciousness and ponder them loosely, without getting into much technical detail.

Table of contents

1. Intro
2. Layers
3. Do we need Axi?
4. GAS BILLS

Plan of the presentation

- Why languages have layers.
- Do we need Axi?
- The programming layer.
- The logical layer.
- The tactic layer.
- The module layer.
- Language features.

Why English is a bad logical system

This sentence is false.

Is the above sentence true or false? If we assume it is false, then apparently it tells the truth about itself, so it is true... but if we assume it is true, then it lies about itself, so it is false.

This situation is deeply problematic – it demonstrates that the English language doesn't fit into the confines of classical logic. We could try to analyze it as fitting some other kind of logical system, but alas, that would also fail – Gödel's incompleteness theorem put on our philosopher hats and try to escape the problem by

No

Just kidding :)

Code on-chain or off-chain?

? Source code on chain or compiled code on chain?

Some jargon

"Semantics" or "dynamic semantics" refers to the meaning and behaviour of programs at run-time.

"Type system" or "static semantics" refers to the constraints that are automatically checked before running programs.

"Logic" or "proving" refers to the correctness guarantees of programs that need to be proved manually. They are also pre-runtime.

What does a programming language consist of?

- Concrete syntax (what the programmer writes).
- Abstract syntax (the internal representation of what the programmer wrote).
- Operational semantics (how programs are executed).
- Type system (which programs are ok).

Paying for on-chain execution

Execution of programs on-chain needs to be paid for. If it weren't, a malicious user could ask the blockchain to execute some really long-running program, or even worse, a non-terminating program, stalling the chain for long or forever. To prevent this, we require users to pay gas for executing their programs.

Cost semantics

The language's operational semantics tell us how how computation proceeds. We can extend it with **cost semantics** which will also tell us how much that computation costs in terms of some resources, like time, memory or **gas**.

Since computation on-chain costs gas, Axi will need a cost semantics.

Cost semantics – a research opportunity

There's also a research opportunity. Since Axi will have a formal specification and powerful logic built-in, a question arises: can we do anything cool with the cost semantics, like proving upper bounds on gas costs?

Note that the naive approach to this topic doesn't work, because programs that do the same thing (and thus are considered equal by the logic) may nevertheless use different amounts of resources. For example, programs which implement quick sort and selection sort are equal because for all inputs they return the same outputs (there's only one way of correctly sorting a list), but their time complexities are different (and so would be their "gas complexities").

Out-of-gas exception in the operational semantics?

A question arises: do we need to model out-of-gas situations in the language semantics?

- On the one hand, programs written in proof assistants (Coq, Lean, Agda) can encounter plenty of run-time problems, like stack overflows, but usually the operational semantics isn't concerned with them because it represents execution on an idealized computer.
- On the other hand, it would make the semantics more realistic and more in line with the actual implementation of the VM (there are techniques to automatically turn a language semantics into a corresponding abstract machine).

I think it would be preferable to not have to deal with out-of-gas situations in the operational semantics...

No out-of-gas exception in the type system

If we have the out-of-gas exception in the semantics, further questions arise – should we track it at the type system level?

It seems that the out-of-gas exception:

- Cannot be thrown by the programmer, as gas depends on VM internals.
- Cannot be caught by the programmer, as the program can't proceed anyway, because it ran out of gas.

So it seems having the out-of-gas exception at the type system level is pointless, but I might be wrong.

Type checking – on-chain, not off-chain

Before running a program we need to type check it to make sure it doesn't contain obvious errors. A question arises: is the type-checking performed off-chain or on-chain?

Type checking off-chain might seem preferable, because we don't need to pay for it (or rather, it's just much cheaper). However, what guarantees would we have that programs on-chain are well-typed? If type checking happened off-chain, an attacker could execute an ill-typed program on-chain, causing a crash. The only ways to avoid this is either performing type checking during run-time, which means the language effectively becomes dynamically typed, or performing type checking on-chain before execution.

Type checking on-chain – questions

With type checking happening on-chain, even more questions arise.

- Does the user need to pay gas for type checking?
- How much? Maybe the cost can be amortized over the cost of execution?
- What's the time complexity of the type checker?
- Does the type checker need its own cost semantics?

Complexity of (Hindley-Milner) type inference

Hindley-Milner type inference (the foundation for languages like Haskell or OCaml) in the worst case takes time doubly exponential in the size of the program (example: $f_0(x) = (x, x); f_{n+1} = f_n \circ f_n$; if I'm not mistaken, there should be 2^{2^n} type variables in the type of f_n).

So in theory, Hindley-Milner is really bad. But in practice, it works well and fast. What gives?

Type inference vs type checking

Wait a sec, so far I've been mostly talking about type checking, but Hindley-Milner is type inference, not type checking. In case you're missing the difference:

- In type inference, you get the term as input and you need to output its type
- In type checking, you get the term and its type as input and you need to check whether they are a good match.

It's easy to see that type checking can be reduced to type inference: just infer the type and compare it with the input type.

Complexity of type checking

So, since the time complexity of pure type inference is very bad, at least in theory, what's the situation with type checking?

Big table

- STLC without annotations: inference and checking equivalent and PTIME-complete.
- Intrinsic STLC: inference in linear time, so checking too.
- Dual Intrinsic STLC: checking in linear time, inference probably still PTIME-complete.
- System F without annotations: inference and checking equivalent and undecidable.
- System F with annotations: should still be linear time.
- CoC without annotations: checking and inference undecidable.
- CoC with enough annotations: decidable, but arbitrarily complex (we might need to run programs at type level).

Big table'

	Checking	Inference 2	Column 3
STLC without annotations	Data 1	Data 2	Data 3
STLC with annotations	Data 4	Data 5	Data 6

Table: A Simple Table

Paying for type checking

Paying for proof checking