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

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?
- What's the time complexity of the type checker?
- Does the type checker need its own cost semantics?

Paying for type checking

Since type checking happens on-chain and requires computational resources (mostly time), having to pay for it seems like a very reasonable default.

However, I have a really bad feeling about this. It's hard to even imagine what kind of demon it would summon, especially when faced by non FP-savvy programmers.

Arguments against paid type checking

- Possible perverse incentives to the programmer. It could be that weaker and less informative types are cheaper to type check, which would result in less incentives to follow best practices, which leads to weaker correctness guarantees, and that runs directly counter to the entire point of having formal proofs.
- Possible perverse incentives to us! We have less incentives to find smart solutions, since money can solve all the problems.
- It makes the cost semantics of the language much less useful, becuase execution ceases to be the only cost. It also completely obliterates the research opportunities with respect to the cost semantics, because proving stuff about the cost of the type checking is a completely different kettle of fish from proving stuff about the cost of the program.

Free typing for the masses

I believe that typing must be free or else we are facing a scary and unknown territory full of bad things...

Let's to find out what the complexity of typing is and try to come up with some kind of a solution to this dire problem.

Type inference and type checking

Typing might be split into two related problems, type checking and type inference:

- In type checking, you get the term and its type as input and you need to check whether this term has this type.
- In type inference, you get the term as input and you need to find its type (preferably the best possible type – the *principal type* – if it exists).

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.

Also note that often, "type checking" is used as a synonym to "typing".

Complexity of typing – intro

The problems of type checking and type inference are somewhat fragile to talk about, because even for a particular kind of language (we will consider STLC, System F and CoC), a lot depends on the concrete variant of the language in consideration. Also, nobody talks about it, because usually it doesn't matter.

We will discuss two simple cases: no annotations at all, and "enough" or "all" annotations that we could wish for.

Complexity of typing – no annotations

	Checking	Inference
STLC	P-complete	P-complete
System F	undecidable	undecidable
CoC	undecidable	undecidable

In case there are no annotations whatsoever, the matter is rather simple. A quick search reveals that typing for STLC is P-complete, i.e. can be done in polynomial time, but is as hard as the hardest problems in P. GPT tells me that the actual complexity is $O(n^2)$ and the typical case takes linear time.

However, as soon as we wander into polymorphism (System F), inference becomes undecidable (and so does checking, which is equivalent to it). CoC is even more powerful, so also undecidable.

Complexity of typing – enough annotations

	Checking	Inference
STLC	Linear	Linear
System F	Linear	Linear
CoC	decidable	decidable

With enough annotations, the situation changes drastically – checking and inference for both STLC and System F now take linear time (but note that annotations that guarantee fast checking don't necessarily guarantee fast inference).

The situation for CoC also changes, although with enough annotations both problems become merely decidable – the source of complexity is different than in System F, as we will soon see.

Hindley-Milner

So far we've been talking about problems and giving answers in terms of complexity classes or decidability. Let's see how things look for something more down to earth, like the Hindley-Milner (HM) type inference.

HM is a type system with polymorphism, but weaker than System F. HM polymorphism is prenex and rank-1, i.e. quantifiers are allowed only at the top-level and they cannot be nested. The HM type system is so simple that it admits a complete type inference algorithm that can always find a principal (i.e. best, most general) type. Type checking is performed by comparing the inferred type with the input type.

HM lays at the foundation of languages like Haskell or OCaml. It is old and venerable, but rather outdated.

Complexity of HM type inference

HM type inference in the worst case takes time doubly exponential in the size of the program. For 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. Note that the indexing over n happens at the meta-level, ie. we can't define f in HM, but we can manually define f_n for any n.

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

A possible solution (MP)

- Typing is split into two phases: off-chain and on-chain.
- The off-chain phase can have comfortable type inference, possibly with bad complexity, and other goodies.
- In addition to typing a term t, the off-chain phase also synthesises a new term t', which has the same computational behaviour, but more type annotations.
- The on-chain phase type checks the term t' in linear time.
- Since the on-chain phase takes linear time, we don't charge gas for it, but instead amortize the cost over the execution cost (or maybe not, maybe we can just make it free).

Downsides of the possible solution

- The new term t' can possibly be (doubly) exponentially bigger than the term t (see the Hindley-Milner example).
- Since t' can be exponentially bigger, the on-chain phase can still be exponentially expensive! That makes amortization over the execution cost pointless.
- Even if the size of t' is not a problem, it still looks different from t, which is a bit unsettling – after all, off-chain we wrote t (and proved theorems about t), but on-chain we see t'.

But it can work

- Synthesis must produce a term from which the original term can be easily reconstructed. This probably means that there will need to be two kinds of type annotations, programmer's annotations and autogenerated annotations.
- When deploying code, we show the user the synthesized term, and he must accept.
- Type checking is free and not amortized over execution, but the user pays for storage on-chain. It could mean there are different rates for data and code, since code must be type checked, but data does not.

Paying for on-chain storage – another demon

Just as I had a bad feeling about paying for the type checking, I have a bad feeling about paying for storage of code.

There are more perverse incentives lurking around the corner. For example, minified code takes less space than pristine code, so it is cheaper. Websites often minify their JS code for this reason.

We would probably need some more precautions here, like computing the gas cost of storage based on the abstract syntax of the code instead of the concrete syntax. This still has some perverse incentives, as there are equivalent programs with different ASTs, and the cheaper one doesn't need to be the most clear one, but they are far smaller.

Paying for proof checking

The start point for proof checking is the same as for type checking: it takes time, so it makes sense to charge gas for it.

I argued that paying for type checking is bad, so what about proof checking? This time, I'm actually not sure.

Let's start by establishing a basic fact: given enough annotations, proof checking is decidable (of course it is, otherwise all these proof assistants wouldn't exist in the first place).

Hello, Ackermann

Consider the following function, called the Ackermann function:

- $f(0, n) = n + 1$
- $f(m + 1, 0) = f(m, 1)$
- $f(m + 1, n + 1) = f(n, f(m + 1, n))$

This function, even though it might not look like it, grows really fast, and this fact can be used to prove that it is not primitive recursive, which in turn means it takes a really long time to compute it.

Complexity of proof checking

If our programming language has function types and recursion over the naturals, we can define the Ackermann function.

If our logic has the conversion rule, we can prove that the Ackermann function, for some big input, returns an even bigger output, but that requires actually performing the computation, which takes long.

Therefore, proof checking is decidable (given enough annotations), but not primitive recursive. This means that checking a proof can take a more or less arbitrarily long, but finite time, i.e. it is guaranteed to terminate.