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.

Code on-chain or off-chain?

Source code on chain or compiled code on chain

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 these perverse incentives are much weaker.

What is proof checking?

Proof checking is the following task: we are given a proof e and a proposition P and we need to check whether e is a proof of P.

The details proof checking depend on many circumstances, like the programming language (if we want theorems about programs), the logical system we're using, and how exactly these two parts are connected.

Our setting

To make this section less vague, we will talk about proof checking in a setting similar to Poor Man's Axi, in particular:

- Classical second order logic.
- A purely functional programming language based on the lambda calculus.
- Most importantly, the computational aspect of the programming language is plugged into the logic by means of the **conversion rule**, which allows proving equality of programs on concrete inputs by running them.

Proof checking happens on-chain

I won't belabor this point too much, because the argument is similar as for type checking. In fact, as we will see, throwing proof checking off-chain is even harder, because it would require some serious ZK (or rather, just succinct "proof") technology.

Paying for proof checking?

I argued that paying for type checking is bad, so what about paying for proof checking? Is it also bad? The starting point is the same: it takes time, so it makes sense to charge gas for it.

This time however, I don't have any bad feelings to guide my thinking, so to answer this question, we will need to investigate a few things first:

- When is proof checking decidable?
- What is the complexity of proof checking?
- What is the typical performance of proof checking?

Proof checking should be decidable

Proof checking should be decidable, i.e. there should be a program (called the proof checker) that always performs it in finite time. To see why this is important, let's do some philosophy.

Imagine we're dealing with a logical system in which proof checking is not decidable. Someone approaches us and claims to have proved a theorem. To support his claim, he offers us a "proof", but we think he might want to trick us. Happily, we can make sure by just checking the proof, which takes only infinitely long.

The moral of the story is simple: if we can't tell proof from non-proof, bullshit from non-bullshit, in finite time, then we are not dealing with a proper logical system.

Proof checking can be decidable

The good news is that proof checking can be decidable. Of course it can, otherwise all these proof assistants wouldn't exist in the first place!.

However, decidability doesn't come for free. To secure it, we need to meet two requirements which arise from two distinct aspects that make up proof checking.

A "decomposition" of proof checking

$$\text{proof checking} \approx \text{type checking} + \text{program execution}$$

Let's start with an oversimplification. We can think of proof checking as consisting of two components:

- A type-checking-like component. For example, to check that **both** e_1 e_2 proves $P \wedge Q$, we need to check that e_1 proves P and that e_2 proves Q. This is basically the same story as type checking pairs: to check that (t_1, t_2) has type $A \times B$, we need to check that t_1 has type A and that t_2 has type B. This is the Curry-Howard correspondence at work.
- A program-execution-like component. For example, to check that **refl** 4 proves $2 + 2 = 4$, we need to reduce $2 + 2$ and see that the result is indeed 4.

Requirement 1 – annotations

First, we need enough annotations, just like in the case of type checking. This time, however, there are two kinds of annotations: type annotations (like A in $\forall x : A, P(x)$) and proposition annotations (like P in **left-either** P e).

If there aren't enough annotations, proof checking is undecidable, at least if the logic has quantification over propositions (I'm not sure if that's already the case for first-order logic). This is because quantification over propositions is the Curry-Howard counterpart of polymorphism, and as we have already seen, type checking for System F without annotations is undecidable.

Fortunately, getting annotations right is easy and doesn't pose conceptual difficulties.

Requirement 2 – termination

Second, a much more drastic requirement: all programs must terminate.

This should be rather obvious. If a program doesn't terminate, the proof checker might also not terminate when it has to execute this program!

Ensuring that all programs terminate is a bit harder than managig annotations, but also possible. Ensuring termination is the responsibility of the *termination checker*, but let's leave it for now – we'll explore its possible designs later.

Proof checking is decidable

Having said the above, here's our conclusion: if there's enough annotations and all programs terminate, then proof checking is decidable.

We shouldn't be too happy, however. The problem is solvable, but how quickly, exactly?

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.

Proof checking is not primitive recursive

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 use it to 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, even for a simple language, even though proof checking is decidable, it is not primitive recursive. This means that checking a proof can take arbitrarily long, but it is guaranteed to terminate.

Proof checking is fast in practice

We know the theory, but what about practice? How long does proof checking take in practice? I am too lazy to research this topic, and also it's not something that people pay a lot of attention to, so we're entering a slightly slippery ground.

In general, proof checking is very fast and is certainly not the bottleneck of anything. This is because typical proofs don't do a lot of computation. The prototypical case is a proof of a function's correctness, which is a universal statement proved by induction.

The amount of computation in such a proof is scant – every time we do something, like using a lemma, maybe there's an opportunity to perform a single step of computation, like rewriting the function's defining equation and that's it.

Decomposing the solution

Recall the (oversimplified) decomposition of our problem:

$$\text{proof checking} \approx \text{type checking} + \text{program execution}$$

I argued that paying for type checking is bad and I feel similarly about the Curry-Howard analog of it in the proof checking world. Therefore, I think it would be nice to make this part free. Fortunately, it wouldn't be terribly difficult – we can use the same trick as for type checking.

Therefore, we only need to think through the execution part of proof checking.

Four approaches

I see four possible approaches. The main distinction is between those that eschew the conversion rule (and computation during proof checking) and those that don't:

- ("Naive" solution) Keep the conversion rule. The programmer must pay for proof checking.
- ("Paranoid" solution) Remove the conversion rule. The programmer must pay for proof checking.
- ("Optimal" solution) Keep the conversion rule and provide a succinct/ZK "proof" system for proofs of execution. Proof checking is free because it takes linear time.
- ("Realistic" solution) Remove the conversion rule. Proof checking is free because it takes linear time.

What is termination checking?

Termination checking is the following task: we are given the definition of a well-typed function $f : A \to B$ and we need to tell whether it terminates or not.

Why should we care?

We already know that all functions must terminate for proof checking to be decidable. However, this isn't termination checking's raison d'être.

Neither has it anything to do with gas, or even what happens at runtime at all. After all, the universe is going to last a finite amount of time, so all functions will eventually terminate, right? Wrong.

Termination is first and foremost a logical notion. It's evil twin, non-termination, manifests itself most strikingly not at runtime, where it cannot be observed at all (because it would take forever), but in the logic –

Termination checking is undecidable

In general, termination checking is undecidable – it means literally solving the Halting Problem! That's hard!