Type Soundness

1 Types should make sense

Last time we saw a typed language, that is annotated lambda calculus terms and simple types. We talked about how the type side worked: proof trees and such. But we didn't really talk about why this type system is a "good" or "accurate" reflection of the underlying language. What should it do?

Formally, we have two rewrite systems. One for the language, and one for judgements. Naturally, these should be aligned somehow. The classic statement is **well-typed programs never go wrong**. Critically the point is that types should be predictors of runtime behavior. The most general formulation is

Theorem 1 (Soundness)**.** *For a term e_1 and type τ, if $\vdash e_1 : \tau$, if e_1 evaluates to e_2, then e_2 is a value of type τ.*

There are many subtleties here

1. What do we mean by value?
2. Can it do bad things along the way?
3. How would we prove this?

2 Values, and the art of not getting stuck

Let's start by talking about a bit of cheating we did last week. We introduced an N type, and added terms $0, 1, 2, ...$, and defined the annotated lambda calculus. We defined the simple types, and a judgement relation. Then we defined the STLC to be the **terms of the annotated lambda calculus that are well-typed**. We know the annotated lambda calculus is confluent with its β rule. But that doesn't tell us a lot about STLC. So consider some well typed e_1 and e_2 with $e_1 \to^* e_2$ in the annotated lambda calculus. Maybe every path from e_1 to e_2 goes through something not-well-typed. So we might end up with a bunch of *spurious* normal forms!

$$\underbrace{e_1}_{\vdash e_1:\tau} \to ... \to \underbrace{e_i}_{\vdash e_i:\tau} \to \underbrace{e_{i+1}}_{\text{no valid type}} \to ... \to \underbrace{e_n}_{\vdash e_n:\tau}$$

If all such paths look like this, then e_i is a normal form in STLC, and not in untyped lambda. So now we have to worry about normal in what language? We could resolve this by requiring something like

$$\text{normal in STLC} \;\Rightarrow\; \text{normal in annotated lambda}$$

But there are many nonsense normal forms, like

$$4\,x$$

which is a bad thing to return to a user since 4 is not a function. So in general, We need something stronger. The way we rule these out is to designate some expressions as values. These are things we expect to find at the end of a computation. In our language, the values are

1. the literals 1,2,...

2. abstractions $\lambda x : \tau.\, e$

Basically we want no lingering applications. Annoyingly, these might not be normal since the body of the lambda may have further evaluations.

So what we want is more like

$$\text{normal in STLC} \;\Rightarrow\; \text{value in annotated lambda}$$

This is almost the classical definition of progress. We will end up refactoring things to make some inductions a bit nicer.

3 Making sure things don't get worse

We snuck in a hint in the last section. Why should we expect the success terms to have the same types

$$\underbrace{e_1}_{\vdash e_1:\tau} \to ... \to \underbrace{e_i}_{\vdash e_i:\tau} \to ...$$

A priori? there is no reason. But if we want the type to predict runtype behavior, it's extremely desirable that partial execution should not change the types! Formally, we want evaluation to preserve type information, so

$$\vdash e_1 : \tau \wedge e_1 \to e_2 \Rightarrow \vdash e_2 : \tau$$

This just says that the judgements respect the evaluation rules, and conversely that evaluation can never change a type.

Recall the candidate *don't get stuck* rule was

$$\text{normal in STLC} \;\Rightarrow\; \text{value in annotated lambda}$$

which unpacks to

$$\vdash e_1 : \tau \wedge e_1 \not\to e_2 \text{ with } \vdash e_2 : \tau' \Longrightarrow e_1 \text{ is a value}$$

and then to

$$\vdash e_1 : \tau \Longrightarrow e_1 \to e_2 \text{ with } \vdash e_2 : \tau' \;\vee\; e_1 \text{ is a value}$$

but in a world with progress, we don't need the extra condition on e_2. So we can reduce this to

$$\vdash e_1 : \tau \Longrightarrow e_1 \to e_2 \;\vee\; e_1 \text{ is a value}$$

which is progress.

4 Putting it together

So we've now strayed a bit from our goal. What do progress and preservation have to do with the soundness definition we gave earlier? Well, soundness isn't always easy to prove by a direct structural induction. But progress and preservation often are easy. And

$$\text{progress} + \text{preservation} \Rightarrow \text{soundness}$$

The proof is dead simple. Suppose

$$e_1 \to ... \to e_2$$

where $\vdash e_1 : \tau$ and e_2 is normal. Then by induction on the length of this sequence and preservation, $\vdash e_2 : \tau$. Now we can apply progress and normality of e_2 to see that it must be a value.

5 Preservation for STLC

Let's show that STLC has preservation. Last week we already saw that

$$\vdash (\lambda x : \tau'.\, e_1)e_2 : \tau \Longrightarrow e_1[x := e_2] : \tau$$

This is almost preservation, but the full β, can apply anywhere. Not just for terms that are applications at the top level. So we induct, but we do need to strength the induction hypothesis a bit to include a context. Suppose $e \to e'$ with $\Gamma \vdash e : \tau$.

1. Case e is a variable or a literal: there are no rewrite rules that apply
2. Case $e = e_1\, e_2$ with redux in e_1: then $\Gamma \vdash e_1 : \tau' \to \tau$, $\Gamma \vdash e_2 : \tau'$, and $e' = e_1'\, e_2$. By induction, $\Gamma \vdash e_1' : \tau' \to \tau$ as well. $\Gamma \vdash e_1' e_2 : \tau$, so we're happy.
3. Case $e = e_1\, e_2$ with redux in e_2: then $\Gamma \vdash e_1 : \tau' \to \tau$, $\Gamma \vdash e_2 : \tau'$, and $e' = e_1\, e_2'$. By induction, $\Gamma \vdash e_2' : \tau'$. $\vdash e_1' e_2 : \tau$, so we're happy again.
4. Case $e = \lambda x : \tau'.\, e_1$: then $(x, \tau') + \Gamma \vdash e_1 : \tau$, and $e' = \lambda x : \tau'.\, e_1'$ By induction, $(x, \tau') + \Gamma \vdash e_1' : \tau$. Therefore $\Gamma \vdash \lambda x : \tau'.\, e_1' : \tau$.

A proof of this structure is quite rote. And in a language like STLC, it's much easier to prove this than soundness directly. I'm not gonna do progress here, the proof is not hard though.