STLC + Recursive Types

Now we would like to talk about recursive types, and talk about how to prove type safety for STLC + recursive types, starting with a proof for type safety for just STLC.

Well typed programs can't go wrong!

not transitive closure

Type safety If $\cdot \vdash e : \tau$ then $\forall e'. e \mapsto^* e'$
$\Rightarrow \mathrm{val}(e')$
$\vee \ \exists e''. e' \mapsto e''$

Q: Do we formally care that the value we get has the correct type?
A: Good question! Right now, we don't!
While most of you have seen type safety proved using progress and preservation, these are only means to ends: THIS is the theorem we really care about.

"not getting stuck"

Remark: Type safety does not imply strong normalization

$e' \neq e$ because there is no $*$

Progress If $\cdot \vdash e : \tau$ then $\exists e'. e \mapsto e'$ or $\mathrm{val}(e)$

Preservation $\cdot \vdash e : \tau \wedge e \mapsto e' \Rightarrow \cdot \vdash e' : \tau$

If you were allowed to transition to the same term, that's OK!
You just me able to use the operational semantics to take one step. Even if you get to the same term, you're still making progress.
C: But it seems you are not doing anything useful.
A: Well, you are still progressing, just not doing anything useful.
All we really want to avoid is stuckness.

It used to be the case, whenever you proved type safety (in the eighties) people normally did this using denotational semantics: give a mathematical denotations for everything and show the typing rules were sound. Then Wright/Felleisen showed how to do type safety in a syntactic manner. If you prove these two lemmas, you can conclude type safety, by repeated application.

Q: Does this mean if you have any language that satisfies preservation, you can trivially make it progress by adding a looping operation?
A: Yeah! It's not very useful, but there is always a rule that will let it loop. It's type safe! It's not stuck. (laugh)
C: Well, that's because yesterday we were talking about normalization.
A: An important point is that I'm going to work with a language which does NOT have normalization.

You do not <u>need</u> to use progress and preservation to prove type safety.

Recall that yesterday, we defined a predicate, such that if an expression was a member of our predicate, it was strongly normalizing. We'll do the same thing today, but for type safety. Take this simple predicate:

$$\mathrm{Safe}(e) \overset{def}{=} \forall e'.\, e \overset{*}{\longmapsto} e' \Rightarrow \mathrm{val}(e') \text{ or } \exists e''.\, e' \longmapsto e''$$

Q: How does the denotational semantics proof go?
A: You normally have to prove a theorem called Adequacy which shows that the denotational semantics reflect the operational semantics of your language.

The inability to prove adequacy of domain theory for PCF was a huge pain in the neck for PL researchers for a long time.

We're going to define our new relation (like SN) into a different style:

$$\mathcal{V}[\![\tau]\!] = \{v \mid \ldots\}$$
$$\mathcal{E}[\![\tau]\!] = \{e \mid \ldots\}$$

I.e. what are the "values" of type tau; what are the closed expressions of type tau? Instead of defining a single relation of all the types type tau, we're dividing this into values and expressions. There will be a mutually recursive definition. This logical relation will capture safe; they will imply Safe, but Safe will not necessarily imply this relation.

$$\mathcal{V}[\![\text{bool}]\!] = \{v \mid v = \text{true} \quad \text{OR} \quad v = \text{false}\}$$

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] = \{\lambda x{:}\tau_1.\, e \mid \forall v.\ v \in \mathcal{V}[\![\tau_1]\!] \Rightarrow e[v/x] \in \mathcal{E}[\![\tau_2]\!]\}$$

zero or more

$$\mathcal{E}[\![\tau]\!] = \{e \mid \forall e'.\ e \overset{n}{\longmapsto} e' \wedge \text{irred}(e') \Rightarrow e' \in \mathcal{V}[\![\tau]\!]\}$$

where $\text{irred}(e) \overset{\text{def}}{=} \nexists e'.\ e \longmapsto e'$ (distinct from V)

Ⓐ $\cdot \vdash e : \tau \Rightarrow e \in \mathcal{E}[\![\tau]\!]$

Ⓑ $e \in \mathcal{E}[\![\tau]\!] \Rightarrow \text{safe}(e)$

same structure as yesterday!

Once again, let's do the easy step (B) first.

Suppose $e \overset{n}{\longmapsto} e'$. Show $\text{val}(e')$ or $\exists e''.\ e' \longmapsto e''$.
Suppose $\neg\text{irred}(e')$, then trivially $\exists e''$
And if $\text{irred}(e')$, then e' is value by $e' \in \mathcal{V}[\![\tau]\!]$ $\blacksquare$

Now let's consider (A). As last time, we first need to generalize our hypothesis to work with open terms.

$\mathcal{G}$ is a restatement of $\gamma \vDash \Gamma$

$$\mathcal{G}[\![\cdot]\!] = \{\emptyset\}$$

$$\mathcal{G}[\![\Gamma, x:\tau]\!] = \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge v \in \mathcal{V}[\![\tau]\!]\}$$

$$\Gamma \vDash e : \tau \overset{\text{def}}{=} \forall \gamma \in \mathcal{G}[\![\Gamma]\!] \Rightarrow \gamma(e) \in \mathcal{E}[\![\tau]\!]$$

read <u>semantically</u>

If you give me substitutions that belong to my logical relation, if you close off my expression, it belongs to my logical relation.

Goal $\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$

This theorem is called the "Fundamental Property", since in all logical relation proofs you will have to prove something like this. "If you have a well-typed term, if you close off the free variables with things in the logical relation, the result is in the logical relation. It has also been called the "Basic Lemma." This is the theorem that tells you if your logical relation is sensible. After you prove this, you usually can get the theorem you're interested in. (Or, in the case of program equivalence, you may need to do more work.)

Proof proceeds by induction on the derivation. So you'll have a case for every typing rule, the interesting cases being lambda and application.

I am not going to do this proof, because it looks very similar.

Introducing recursive types

this is a binder

$$\begin{aligned}
\tau &::= \dots \mid \alpha \mid \mu\alpha.\tau \\
e &::= \dots \mid \text{fold}_{\mu\alpha.\tau}\, e \mid \text{unfold}\, e \\
v &::= \dots \mid \text{fold}_{\mu\alpha.\tau}\, v \\
E &::= \dots \mid \text{fold}_{\mu\alpha.\tau}\, E \mid \text{unfold}\, E
\end{aligned}$$

$\mid 1$

$\mid \langle\rangle$

$\mid \langle\rangle$

unit

for technical reasons we need the annotation

$$\text{unfold}\,(\text{fold}_{\mu\alpha.\tau}\ v) \longmapsto v$$

$\boxed{\Gamma \vdash e : \tau}$ ...

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \text{fold}_{\mu\alpha.\tau}\, e : \mu\alpha.\tau}$$

$$\frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \text{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

Haskell programmers may be confused by the terminology here;
fold/unfold correspond to roll/unrold, not the recursive higher-order function.

Brief review:

$$\text{list} = \text{Nil} \mid \text{Cons of int} * \text{list}$$

$$\text{list} = \mu\alpha.\,1 + (\text{int} * \alpha)$$

the position where it is repeating itself

In equirecursive typing, we think of a list as equal to its unfolding.

$$\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$$

In isorecursive typing, we think of this as an isomorphism; thus you need an explicit set of functions to go from unexpanded to expanded views. (They are like "markers").

$$\begin{tikzcd}
\mu\alpha.\tau \arrow[r, bend left, "\text{unfold}"] & \tau[\mu\alpha.\tau/\alpha] \arrow[l, bend left, "\text{fold}"]
\end{tikzcd}$$

This is a very useful typing feature!

In homotopy type theory, equivalence is equality! But you still need to transport...

Previously, omega, the canonical example of a nonterminating expression, did not type check.

$$\Omega = \overbrace{(\lambda x.\ x\ x)}^{SA}\ (\lambda x.\ x\ x)$$

With recursive types, we can now make it type check.

$$SA : (\mu\alpha.\alpha \to \tau) \to \tau$$

the strict positivity condition would prevent us from writing this recursive type, keeping Coq sound.

$$SA = \lambda x : \mu\alpha.\alpha \to \tau.\ \underbrace{(\text{unfold } x)}_{(\mu\alpha \to \tau) \to \tau}\ \underbrace{x}_{\mu\alpha.\alpha \to \tau}$$

it type-checks!

$$\text{so}\quad \Omega = \underbrace{SA}_{(\mu\alpha.\alpha\to\tau)\to\tau}\ \underbrace{(\text{fold } SA)}_{\mu\alpha.\alpha\to\tau}$$

We did not bother with the monotonicity conditions; we don't need anything to do both contravariant and covariant types.

This is because we don't have subtyping.

Let's now extend our logical relation with recursive types.

$$V[\![\text{bool}]\!] = \{v \mid v = \text{true} \quad \text{OR} \quad v = \text{false}\}$$

$$V[\![\tau_1 \to \tau_2]\!] = \{\lambda x{:}\tau_1.e \mid \forall v.\ v \in V[\![\tau_1]\!] \Rightarrow e[v/x] \in \mathcal{E}[\![\tau_2]\!]\}$$

$$V[\![\mu\alpha.\tau]\!] = \{\text{fold}_{\mu\alpha.\tau}\ v \mid \underbrace{v \in V[\![\tau[\mu\alpha.\tau/\alpha]]\!]}_{\text{uh oh! this is no longer well-founded}}\}$$

We tried to use a bigger type in our definition of the mu-relation. How do we know this fixed point exists? We don't. How do we solve this problem?

Step-indexed logical relations!

$$V_k[\![\tau]\!] = \{v \mid \dots\}$$

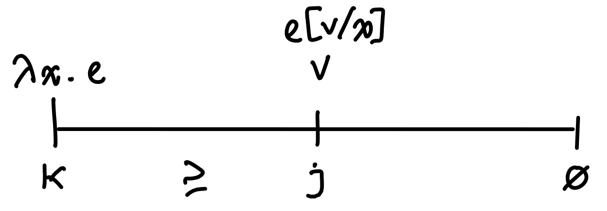
A value is a member of this relation, if it looks like it's type tau for k steps. Afterwards, we don't say anything about it! Now our definition works on the number of steps, as well as type tau, a double induction.

$$V_k[\![\text{bool}]\!] = \{v \mid v = \text{true} \quad \text{OR} \quad v = \text{false}\}$$

allowed to be the same?

$$V_k[\![\tau_1 \to \tau_2]\!] = \{\lambda x{:}\tau_1.e \mid j \le k.\ \forall v.\ v \in V_j[\![\tau_1]\!] \Rightarrow e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\}$$

We may not evaluate the lambda right away. We'll use up some number of steps, and when we have j steps left, we might apply the lambda. But at this point, we better apply it to an argument which is good for the remaining j steps.



Why not j-1? After all, actually doing e[v/x] takes a step. Well it turns out that I don't really need the v to be good for one extra step.

$$\underbrace{(\lambda x.x)\ v}_{j+1} \longmapsto \underbrace{v}_{j}$$

Intuitively, v is unaffected by the substitution. (It's clear in the proof itself.)

Alternate (equivalent) definition:

$j < k \qquad j \qquad j+1$

$$\mathcal{V}_k[\![\mu\alpha.\tau]\!] = \{\mathrm{fold}_{\mu\alpha.\tau}\, v \mid \forall j \overset{\text{natural numbers}}{<} k.\ v \in \mathcal{V}_{\underset{\text{originally } k-1}{j}}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\}$$

The idea here is that the only way to get to the inside of a fold is an unfold, which would take up a step. Thus, we don't need the v for very many steps.

Q: What is V_0?
A: It contains everything. But this isn't the interesting question: what is E_0?
Q: Well, I was wondering what happens with k-1
A: Ah, let me generalize this definition [shown above].

Now we're going to work on E. E has the type tau for k-steps. So what do you do with E? Well, you run it. But recall you have a bound; you can't run it for more than k steps.

$$\mathcal{E}_k[\![\tau]\!] = \{e \mid \forall j<k.\ \forall e'.\ e \xmapsto{j} e' \wedge \mathrm{irred}(e') \Rightarrow e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

Q: Why is j strictly less than k?
A: Well, with step-indexed relations, the indices can be somewhat fiddly. I've placed it on E instead of the function case. The time is going down, so you're allowed to use it. If we ate up k steps, we would have zero steps left. I suppose that would work, and on my feet I can't think of it, but it's not really meaningful to be in V for zero steps. Let me think about it.

$$\mathcal{G}_k[\![\cdot]\!] = \{\emptyset\}$$

$$\mathcal{G}_k[\![\Gamma, x:\tau]\!] = \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}_k[\![\Gamma]\!] \wedge v \in \mathcal{V}_k[\![\tau]\!]\}$$

$$\Gamma \vDash e : \tau \overset{\mathrm{def}}{=} \forall k \geq 0.\ \forall \gamma \in \mathcal{G}_k[\![\Gamma]\!] \Rightarrow \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

Q: The step-indexes have polluted everything. Is there any salvation?
A: No. Note that with step-indices, you can immediately tell that it's a well-founded relation; these sets can't be uninhabited. If you weren't using step-indexes, you would have to write things that weren't obviously founded. In those cases, you have to do a lot of work to show the relation is inhabited. People have done this sort of stuff. What this is saving you is a lot of extra mathematical work to prove the relation is sensible. The PRICE is the ks are now polluting everything.

It is a bit like dependent types.

Note that some people have come up with ways to work around the ugliness of the indices.

So we could write down an operational semantics where they are all zero-step operations, and only fold/unfold is a one-step operation. So the intuition is nice in our case, and you can make it nicer but now you have to specialize. You've just defined a metric.

Important note: I said some expression looks like tau up to k step. So what if you had fewer steps? Yes! If you have more, it shouldn't. What I'm trying to get at, is that these logical relations rely on a downward closure/monotonicity. So this is an important lemma you have to prove:

Lemma Downward closure/Monotonicity

If $j \leq k$ and $v \in V_k[\![\tau]\!]$ then $v \in V_j[\![\tau]\!]$

Proof by induction on type tau. Here's the mu case:

Have $^{*}\text{fold } v \in V_k[\![\mu\alpha.\tau]\!]$

Show $\text{fold } v \in V_j[\![\mu\alpha.\tau]\!]$

Suppose $j' < j$. Show $v \in V_{j'}[\![\tau[\mu\alpha.\tau/\alpha]]\!]$

Instantiate $*$ with $j' < j \leq k \therefore v \in V_{j'}[\![\tau[\mu\alpha.\tau/\alpha]]\!]$

Exercise: do the function case. Notice that we carefully said less-than-or-equal to k. If we did not talk about an arbitrary j and just put in k, there will be a problem. This is incredibly important! You need to talk about the future.

Theorem (Fundamental Property): $\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$

Exercise: do the lambda case and the function case. You will need to be careful about decrementing the steps; you will sometimes have too many steps, in which case the monotonicity lemma comes in handy. In other cases, you'll have sequence of reductions and decrement them appropriately.

Q: Last time, you went to logical relations because you needed a strengthened induction hypothesis. But here, if you did a progress/preservation, you wouldn't need a strengthened induction hypothesis.

A: Yes, that would work fine. And here we have to do step-indexed. But we're building towards program equivalence, at which point you do NEED step-indexed logical relations for recursive types. My PhD types was about step-indexed logical relations for mutable references. When I finished, I thought this was lame, because progress/preservation could always done. Program equivalence is the KILLER APP.

Q: Does CBN work the same way?
A: Yes, so long as you stay faithful to your operational semantics. You have to tweak the definitions based on the language semantics.