Today we're going to start talking about binary relations. The canonical example is equivalence, e.g. given

$$e_1 = (\lambda f{:}\mathsf{int}\to\mathsf{int}.\ f\ 5)\ (\lambda x{:}\mathsf{int}.\ x)$$
$$e_2 = (\lambda x{:}\mathsf{bool}.\ 5)\ \mathsf{true}$$

It should be pretty easy to see that these programs are "equal" in some case. Our goal is to build a more powerful framework for proving these kinds of equivalences.

What is equivalence? Well, the standard definition is called "contextual equivalence" (or "observational equivalence"), which says that for all possible program contexts, when you put some expression in it, it will "behave" the same way as the other context.

this is defined below

Contextual equivalence: $\Gamma \vdash e_1 \approx^{ctx} e_2 : \tau = \forall C : (\Gamma \vdash \tau) \rightsquigarrow (\cdot \vdash \mathsf{bool})$

a program should not have FVs

We should first say a little bit about what we mean by "context". In the simply typed lambda calculus, we can think of contexts as expressions with holes in them.

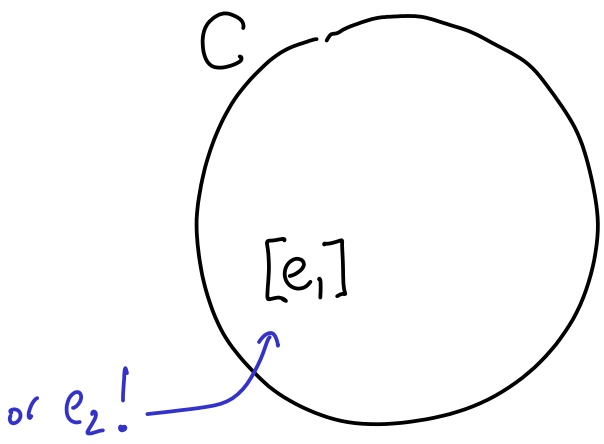
$$\begin{aligned} C ::= {} & [\cdot] \\ \mid {} & \text{if } C \text{ then } e_1 \text{ else } e_2 \\ \mid {} & \text{if } e \text{ then } C \text{ else } e_2 \\ \mid {} & \text{if } e \text{ then } e_1 \text{ else } C \\ \mid {} & \lambda x.\ C \\ \mid {} & C\ e \\ \mid {} & e\ C \end{aligned}$$

"hole"

What type do you need here? You only need a base type inhabited by TWO elements; this is enough! And because we're quantifying over all contexts, nothing is lost! (But unit doesn't work)

this makes general contexts different from evaluation contexts, since we never evaluate under the lambda

I want to say the context is a big bubble with a hole in it, and when I stick e1 or e2 in the hole, I should get the same thing out.



However, we don't want to just talk about untyped contexts; we want our contexts to be "well-typed".

$$C : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$$

type of the hole

type of the resulting filled in context

$\Gamma \vdash e : \tau \rightsquigarrow \Gamma' \vdash C[e] : \tau'$

The typing rules for contexts are fairly simple, but we'll do a few to get the feeling:

this was incorrect in lecture ->

$$\frac{\Gamma' \subseteq \Gamma}{\vdash [\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau)}$$

our SLTC allows weakening so we build it into the rule

clearly it will be the same type

Q: Don't you not want to allow wanting weakening here? Because whenever you calculate the term, you could do it after the fact.
A: For our construction of contextual equivalence, omitting weakening here can cause problems for us. [someone in the audience explained but I missed the explanation]

these are the traditional return types

$$\frac{\vdash C : (\Gamma, x:\tau_1 \vdash \tau) \rightsquigarrow (\Gamma', x:\tau_1 \vdash \tau_2)}{\vdash \lambda x:\tau_1.\, C : (\Gamma, x:\tau_1 \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau_1 \to \tau_2)}$$

notice the hole could be buried arbitrarily deep, so we ignore the τ

Q: What about multiple holes?
A: There is a theorem that lets you just generalize this to multiple holes. Saying one hole keeps things simpler.

Aside: In ATTAPL, Andrew Pitts defines contextual equivalence as a greatest congruence relation. It deals with a language with polymorphism and existential types, but only allows recursive types and doesn't use step-indexed LRs (it predates it); he uses Galois connections to show recursive functions satisfies admissibility conditions. At the end of the chapter, he says, it can handle recursive problems, but open problem: how do we scale it up for recursive types? Well, we have that machinery!

Point of confusion: the judgment we are using to indicate in holes is reminiscent of the typing rules from type theory (e.g. t2 is a type under some context). So it looks like there is a dependence, but this is false: we're implicitly referring to the expression we're going to fill the hole with.

Q: But what about termination?
A: Well, at the moment, we're in STLC, so everythign terminates. Note that we cannot pick a function as our base type, because it's not at all clear to say they are equal. Note that you can always build a bigger context which will distinguish them, but eventually, you want to stop somewhere where equality is purely syntactic. Notice that it is not all clear how you would go about proving their equivalent in ALL contexts (the only thing you could possibly try is induction on C, and the lambda rule will get you stuck): that's why we're using logical relations! (But another popular method is bisimulations.)

It's well worth noting that bisimulation is a form of coinduction; (co)induction really is what makes the world go around!

Interesting fact: when nontermination is admitted in the language, you don't really care about the return type: the only thing you care about is whether or not they coterminate or not!

Contextual equivalence for non-normalizing languages:

$$\Gamma \vdash e_1 \approx^{ctx} e_2 : \tau = \forall \tau'.\, \forall C : (\Gamma \vdash \tau) \rightsquigarrow (\cdot \vdash \tau')$$

just check termination

But we are not going to use this definition today...

Q: What if I have other effects?
A: It still works!
Q: I don't believe you. For example, compare a program which prints and then runs forever, and a program which just runs forever?
A: Well, you need some sort of construct which could see what was printed... in the example of references, you can always read out the reference to see what was printed.

A brief application: imagine that you are implementing an abstract data structure, e.g. a stack. You have an existential data type, so the internals are hidden, but you would like to show your two implementations are behaviourally equivalent.

System F

a "suspended" computation (a polymorphic function)

$$\tau ::= \text{bool} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha.\tau \mid \exists \alpha.\tau$$

$$e ::= \ldots \mid \Lambda\alpha.e \mid e[\tau] \mid \text{pack}\langle\tau', e\rangle \text{ as } \exists\alpha.\tau \mid \text{unpack } \alpha, x = e_1 \text{ in } e_2$$

included for technical reasons

$$v ::= \Lambda\alpha.e \mid \text{pack}\langle\tau, v\rangle \text{ as } \exists\alpha.\tau$$

$$E ::= \ldots \mid E[\tau] \mid \text{pack}\langle\tau, E\rangle \text{ as } \exists\alpha.\tau \mid \text{unpack } \alpha, x = E \text{ in } e$$

$$(\Lambda\alpha.e)[\tau] \longmapsto e[\tau/\alpha]$$

$$\text{unpack } \alpha, x = (\text{pack}\langle\tau, v\rangle) \text{ in } e \longmapsto e[\tau/\alpha][v/x]$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$ $\quad \Delta ::= \cdot \mid \Delta, \alpha \quad \Gamma ::= \cdot \mid \Gamma, x : \tau$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau \quad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash e : \tau[\tau'/\alpha] \quad \Delta \vdash \tau'}{\Delta; \Gamma \vdash \text{pack}\langle\tau', e\rangle \text{ as } \exists\alpha.\tau : \exists\alpha.\tau}$$

does not contain α's; prevents α from leaking outside

$$\frac{\Delta; \Gamma \vdash e_1 : \exists\alpha.\tau \quad \Delta, \alpha; \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash \text{unpack } \alpha, x = e_1 \text{ in } e_2 : \tau_2}$$

Many presentations of System F omit existential types, because they are not strictly necessary; they are included here because they are very interesting rule in the logical relation.

There are a lot of cool things you can do with polymorphism and existential types. For example, Wadler conceived of the notion of a "free theorem". Here is an example:

$$\cdot \vdash e : \forall \alpha. \alpha \to \alpha \quad \wedge \quad \cdot \vdash \tau \quad \wedge \quad \cdot \vdash v : \tau$$
$$\Rightarrow \quad e[\tau]\ v \mapsto^{*} v$$

i.e. the implementation of e must be $\Lambda \alpha. \lambda x{:}\alpha . x$

Why is this true? Well, it has to do with the power of the type. e has stated in its type that it is willing to accept any type (even if it was empty). This is such a powerful restriction on its behavior that it fully specifies its type. Note that if you have nontermination, you have to weaken this a little:

$$\Rightarrow \quad e[\tau]\ v \mapsto^{*} v \quad \text{OR} \quad e[\tau]\ v \Uparrow$$

It's interesting to see how you could actually prove this theorem with logical relations, though we will probably not get to it today.

Here is an example of using an existential type.

$$\tau = \exists \alpha.\ \alpha * (\alpha \to \text{bool})$$
$$e_1 = \text{pack} \langle \text{int}, (4, \lambda x{:}\,\text{int}.\ x \overset{\text{int}}{=} 0) \rangle$$
$$e_2 = \text{pack} \langle \text{bool}, (\text{true}, \lambda x{:}\,\text{bool}.\ \text{not}\ x) \rangle$$

These expressions look different, but amazingly they are equivalent! Alpha is intended to be an opaque blob, and when we use the function wrapped up the existential, the only thing we can ever use it with is the other value passed with it. So both of these functions always return false.

Q: So you have no decidable equality between types?
A: Yes, I do not have intensional type analysis in this language.
Q: What if I type e1, and put forth in some variable x, and then use it in some other term? I can discriminate values in that case?
A: The unpack rule will save you. You can't take that for and give it out to the world.

Time for some logical relations. Recall last time we had:

$$V[\![\text{bool}]\!] = \{ v \mid v = \text{true} \vee v = \text{false} \}$$
$$\mathcal{E}[\![\tau]\!] = \{ e \mid \cdots \}$$

Now we want to construct binary relations instead of unary relations, and we want our relations to capture what it means for values to be observationally equivalent.

$$\text{Atom}[\tau] = \{ (e_1, e_2) \mid \cdot \vdash e_1 : \tau \wedge \cdot \vdash e_2 \tau \}$$

a little bit of formalism here, because I want to make sure my terms have the right type (and because this will become nontrivial later)

$$V[\![\text{bool}]\!] = \{ (v_1, v_2) \in \text{Atom}[\text{bool}] \mid v_1 = v_2 = \text{true} \vee v_1 = v_2 = \text{false} \}$$ obvious!

$$V[\![\tau_1 \to \tau_2]\!] = \{ (\lambda x{:}\tau_1.\, e_1, \lambda x{:}\tau_1.\, e_2) \in \text{Atom}[\tau_1 \to \tau_2] \mid \forall (v_1, v_2) \in V[\![\tau_1]\!].\ (e_1[v_1/x], e_2[v_2/x]) \in \mathcal{E}[\![\tau_2]\!] \}$$

$$\mathcal{E}[\![\tau]\!] = \{ (e_1, e_2) \in \text{Atom}[\tau] \mid \exists v_1, v_2.\ e_1 \mapsto^* v_1 \wedge e_2 \mapsto^* v_2 \wedge (v_1, v_2) \in V[\![\tau]\!] \}$$

alternately $\forall v_1.\ e_1 \mapsto^* v_1 \Rightarrow \exists v_2.\ e_2 \mapsto^* v_2 \wedge (v_1, v_2) \in V[\![\tau]\!] \}$

Additional caveat: When you are non-terminating, this star will be an explicit number of steps, since we will be using step-indexing.

note that this symmetric relation only works because our language is terminating; when we have a nonterminating language we are more likely to use the asymmetric relation. In fact, we will often have a notion of "contextual approximation"

$$\Gamma \vdash e_1 \le^{ctx} e_2 : \tau \stackrel{def}{=} \forall C : \cdots \Rightarrow C[e_1]\Downarrow \Rightarrow C[e_2]\Downarrow$$

$$\Gamma \vdash e_1 \approx^{ctx} e_2 : \tau \stackrel{def}{=} \Gamma \vdash e_1 \le^{ctx} e_2 \wedge \Gamma \vdash e_2 \le^{ctx} e_1$$

This makes things easier, because we cut our work in half, and then can often say "without loss of generality" for the other direction.

We now consider polymorphic types, which are inhabited by type lambdas. We might attempt to approach these by analogy to lambdas, where we applied them to related values. What are "related types"? Well, we might think that it doesn't matter, after all, the type has no bearing on the eventual "result" of the type-lambda. Consider this example:

$$(\Lambda\alpha.\lambda x{:}\alpha.\ 5)[\text{bool}]\ \text{true} \mapsto^* 5$$
$$(\quad\text{—}''\text{—}\quad)[\text{int}]\ 100 \mapsto^* 5 \qquad \text{(Example 1)}$$

Most people would reasonably expect our logical relations to be able to say these two programs are equivalent. So we'll have to allow the TYPES that we apply the type lambdas to be different. Let's give it a try:

$$\mathcal{V}[\![\forall\alpha.\tau]\!] = \{(\Lambda\alpha.e_1, \Lambda\alpha.e_2) \in \text{Atom}[\forall\alpha.\tau]$$
$$\text{Attempt:}\quad \mid \forall\tau_1,\tau_2.\ (\underbrace{e_1[\tau_1/\alpha]}_{\tau[\tau_1/\alpha]}, \underbrace{e_2[\tau_2/\alpha]}_{\tau[\tau_2/\alpha]}) \in \mathcal{E}[\![\tau]\!]\ \}$$

Unfortunately, we now have a problem. How are we supposed to relate our two expressions? The type application caused the expressions to get their concrete types substituted in for their type variables, but for our choice of the expression relation, we still have a type variable alpha free in tau, and no indication whether or not we should pick t1 or t2. What we would like is to defer the choice, and GENERALIZE our relation to work over OPEN types.

We'll generalize our type relation by introducing a type environment rho which we can use to keep track of the assignments to type variables. This will not be a simple map: we'll need to track both the assignments for the left hand side, AND the right hand side. For convenience, we'll introduce some notation:

ρ maps a type variable to types (and a relation)

To refer to a specific type (e.g. the mapping from type variables to the lhs type, we'll have syntax for projections which give you the map for a specific index, as shown here:

$$\rho = \{\alpha \mapsto (\tau_1, \tau_2, R), \cdots\}$$
$$\rho_1 = \{\alpha \mapsto \tau_1, \cdots\}$$
$$\rho_2 = \{\alpha \mapsto \tau_2, \cdots\}$$
$$\rho_R = \{\alpha \mapsto R, \cdots\}$$

We will juxtapose rho with a type to apply it as a substitution. So, for example:

$$\rho_1(\alpha \to \alpha) \stackrel{\text{def}}{=} \tau_1 \to \tau_1$$

ignore the green R for now, it will be motivated later...

Just as was the case with step indexing, we'll need to thread rho through all of our definitions. For the four definitions we already created, we expect things to be straightforward: this is almost true. The most nontrivial change will be for atom. Remember that atom is generating a set of pairs of expressions which are of "type tau". Tau may have free type variables, which are going to need to be instantiated differently on the lhs and rhs. We're going to define Atom by way of an intermediate step, which will help us later.

$$\mathrm{Atom}[\tau_1, \tau_2] = \{(e_1, e_2) \mid \cdot \vdash e_1 : \tau_1 \wedge \cdot \vdash e_2 : \tau_2\}$$

$$\mathrm{Atom}[\tau]\,\rho \stackrel{\text{def}}{=} \mathrm{Atom}[\rho_1\tau_1, \rho_2\tau_2]$$

As you might expect, e1 and e2 may not have the same type! They will only be "related."

Warning: In lecture the second definition was defined directly as:

$$\mathrm{Atom}[\tau]\,\rho = \{(e_1, e_2) \mid \cdot \vdash e_1 : \rho_1\tau_1 \wedge \cdot \vdash e_2 : \rho_2\tau_2\}$$

The bool case is trivial.

$$\mathcal{V}[\![\mathrm{bool}]\!]\,\rho = \{(v_1, v_2) \in \mathrm{Atom}[\mathrm{bool}]\,\rho \mid v_1 = v_2 = \mathrm{true} \;\vee\; v_1 = v_2 = \mathrm{false}\}$$

The function case proceeds as before, but we have to be a little careful when we state the structure of the elements we extract out of atom: the types of the binders are not t1 and t2, but rho1(t1) and rho2(t2) (remember substitutions affect the types of binders too.)

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!]\,\rho = \{(\lambda x{:}\rho_1(\tau_1).\, e_1, \lambda x{:}\rho_2(\tau_1).\, e_2) \in \mathrm{Atom}[\tau_1 \to \tau_2]\,\rho \mid \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]\,\rho .\; (e_1[v_1/x], e_2[v_2/x]) \in \mathcal{E}[\![\tau_2]\!]\,\rho\}$$

$$\mathcal{E}[\![\tau]\!]\,\rho = \{(e_1, e_2) \in \mathrm{Atom}[\tau]\,\rho \mid \exists v_1, v_2 .\; e_1 \mapsto^* v_1 \wedge e_2 \mapsto^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[\![\tau]\!]\,\rho\}$$

Finally, we can return to the case that gave us so much trouble. Instead of performing a substitution on tau right away, we simply add the two choices to our context:

$$\mathcal{V}[\![\forall\alpha.\tau]\!]\,\rho = \{(\Lambda\alpha.e_1, \Lambda\alpha.e_2) \mid \boxed{\forall\tau_1,\tau_2.}\ (e_1[\tau_1/\alpha], e_2[\tau_2/\alpha]) \in \mathcal{E}[\![\tau]\!]\,\rho[\alpha \mapsto (\tau_1,\tau_2)]\}$$

Of course, we will need to use the context somewhere, and this use will come when we use the type variable that the type lambda bound. We should now think about how we're actually going to define this relation. When are two values related at the type alpha?

Remember the opaque blobs had type alpha!

$$\mathcal{V}[\![\alpha]\!]\,\rho = \{(v_1,v_2) \in \mathrm{Atom}[\alpha]\,\rho \mid ???\}$$

Just as we got to pick two types to apply the type lambdas to (displayed in a blue box), we should also be able to pick a relation that specifies when values of these two types t1 and t2 should be related to each other. So we augment our context rho with yet another component, R : Rel[t1, t2] The final, true relation on type lambdas is thus:

(well, not really augment; it was there already ;-)

atoms restricted to values

$$\mathrm{Rel}[\tau_1,\tau_2] = \{R \in \mathcal{P}(\mathrm{Atom}^{\mathrm{val}}[\tau_1,\tau_2])\}$$

power set

Rel[t1, t2] relates a value of type t1 and a value of type t2.

$$\mathcal{V}[\![\forall\alpha.\tau]\!]\,\rho = \{(\Lambda\alpha.e_1, \Lambda\alpha.e_2) \in \mathrm{Atom}[\forall\alpha.\tau]\,\rho \mid \forall\tau_1,\tau_2.\ \forall R \in \mathrm{Rel}[\tau_1,\tau_2].\ (e_1[\tau_1/\alpha], e_2[\tau_2/\alpha]) \in \mathcal{E}[\![\tau]\!]\,\rho[\alpha \mapsto (\tau_1,\tau_2,R)]$$

$$\mathcal{V}[\![\alpha]\!]\,\rho = \{(v_1,v_2) \in \mathrm{Atom}[\alpha]\,\rho \mid (v_1,v_2) \in \rho_R(\alpha)\}$$

In the case of a type lambda, the choice of relation is universally quantified: thus, we only relate type lambdas who are equivalent under ANY choice of relation.

You might think that there would be some trouble if you had a type (forall a. a), since the relation might be empty. But this type is uninhabited, whereas in the case of (forall a. \x:a. x), the condition on their equivalence is vacuously fulfilled!

Finally, we can consider the rule for existential types. Like their universal counterparts, we will need to specify a relation. However, this relation is existentially quantified (instead of universally quantified)

$$\mathcal{V}[\![\exists\alpha.\tau]\!]\,\rho = \{(pack\langle\tau_1,v_1\rangle, pack\langle\tau_2,v_2\rangle) \in Atom[\exists\alpha.\tau]\,\rho \mid \exists R \in Rel[\tau_1,\tau_2].\ (v_1,v_2) \in \mathcal{V}[\![\tau]\!]\,\rho[\alpha \mapsto (\tau_1,\tau_2,R)]\}$$

adding existential quantifiers here for t1 and t2 is unnecessary

Here, the user can give us a specific relation for which he wants the equivalence to hold; in the case of example of existential types, it may include (4, true). We will give more intuition for the function R is playing next lecture.