— Motivating step-indexing

Recall the Støvring and Lassen example from last week.

The problem: Apparently it's unprovable in our current model. So the example motivates our next extension to the model (step-indexing).

Recall the example:

$\tau := \mu\alpha.(\neg\text{int})\longrightarrow\alpha$
$\varphi_1 :=$ fix f(x:τ):int. callcc(k.f((unfold x)k))
$\varphi_2 := \lambda$y:τ.callcc(k.(fix f(x:τ):int. f((unfold x)k)) y).

Prop: $\varphi_1 \equiv \varphi_2 : \tau \longrightarrow$ int.

The point is φ_1 captures a new continuation every time around the loop while φ_2 only captures one continuation. But it doesn't matter.

Last time, we got stuck early. Let's push the proof a little further to get well and truly stuck.

Proof idea:

Let $(K_1,K_2) \in K[\text{int}]$, $(v_1,v_2) \in V[\tau]$.
TS: $K_1[\varphi_1\ v_1]\downarrow\downarrow K_2[\varphi_2\ v_2]$.

First, evaluate these guys to make the structure of the things match up, more or less.

$K_1[\varphi_1\ v_1] \mapsto^* K_1[\varphi_1((\text{unfold } v_1)(\text{cont } K_1))]$
$K_2[\varphi_2\ v_2] \mapsto^* K_2[F\_\{K_2\}\ ((\text{unfold } v_2)\ (\text{cont } K_2))]$

where $F\_\{K_2\} :=$ fix f(x:τ):int. f((unfold x)(cont K_2)).

STS: $K_1[\varphi_1((\text{unfold } v_1)(\text{cont } K_1))] \downarrow\downarrow K_2[F\_\{K_2\}\ ((\text{unfold } v_2)\ (\text{cont } K_2))]$.

Last time, we observed the following pairs are related:

$((K_1,K_2) \in K[\text{int}] \Longrightarrow (\text{cont } K_1,\text{cont } K_2) \in V[\neg\text{int}]) \wedge$
$((v_1,v_2) \in V[\tau] \Longrightarrow (\text{unfold } v_1,\text{unfold } v_2) \in E[(\neg\text{int})\longrightarrow\tau])$.

and claimed STS $(\varphi_1,F\_\{K_2\}) \in V[\tau]$. But to use this argument, you have to know they're equivalent under arbitrary contexts. Clearly not: φ_1 uses callcc while $F\_\{K_2\}$ doesn't—it just assumes its current continuation is K_2 and behaves that way. That idea certainly doesn't work.

We can try something else. Given the pairs above,
WK: $((\text{unfold } v_1)(\text{cont } K_1), (\text{unfold } v_2)(\text{cont } K_2)) \in E[\tau]$.
Thus

† STS: $(K_1[\varphi_1(\bullet)],K_2[F\_\{K_2\}(\bullet)]) \in K[\tau]$.

This is the right way to procede. (We'll be able to complete this proof once we change the model in the way that we'll need.)

Let's try to prove † to see where it breaks down.

Let $(w_1,w_2) \in V[\tau]$.

(1) TS: $K_1[\varphi_1\ w_1] \downarrow\downarrow K_2[F\_\{K_2\}\ w_2]$

Note we've kinda gone back to the beginning.

Reducing both sides,

(2) STS: $K_1[\varphi_1((\text{unfold}\ w_1)(\text{cont}\ K_1)] \downarrow\downarrow K_2[F\_\{K_2\}\ ((\text{unfold}\ w_2)(\text{cont}\ K_2))]$.

But we've reached our starting point.

We could try boiling down (1), treating (2) as a coinduction hypothesis. The following does not work, but provides good intuition:

Show: $\forall n.$
$K_1[\varphi_1((\text{unfold}\ w_1)(\text{cont}\ K_1))] \downarrow\downarrow n\ K_2[F\_\{K_2\}\ ((\text{unfold}\ w_2)(\text{cont}\ K_2))]$.

Where $\downarrow\downarrow n$ ("co-terminates up to n steps") means if the guy on the left terminates within n steps of computation, then the guy on the right terminates.

End proof idea.

If you try to prove this by induction, it ends up not working. In the definition of the LR at arrow types, we don't count these indices. Moreover, there's nothing in the relation on continuations that talks about the number of steps of computation.

This idea of what goes wrong leads us to step-indexing. Induction on the number of steps in the computation justifies the coinductive argument we'd like to write.

The way you see whether or not coinductive reasoning is valid: Is the argument "productive". Do we always have positive movement in the proof. (You don't want to prove something is true given that it's true. You want to take some steps before the recursive calls.)

The basic idea of step-indexing is to build this induction on the number of steps of computation into the model. (All things are related for some number of steps of computation.)

In retrospect, it makes sense that this example doesn't work. It comes from a paper on bisimulations (and coinductive proof).

Not only is there no proof that these are not equivalent in this language, but Derek is pretty sure they are logically related. They're contextually equivalent and via $\top\top$-closure that should imply logically related.

— A simpler example

Assume
$\vdash e_0 : \text{bool}$
$k : \neg\tau \vdash e_1 : \tau$
$k : \neg\tau \vdash e_2 : \tau$
Show
$\vdash$ callcc(k.if e_0 then e_1 else e_2)
$\equiv$
if e_0 then callcc(k.e_1) else callcc(k.e_2)
$: \tau$

Intuition: In the former, you grab the continuation then do the if; either it goes to e_1 or it goes to e_2. In the latter, you do the callcc before the e_1 or the e_2 but at those points, its the same continuation as in the former example.

HW: Prove this using our model from last time (and parametricity).

Aside: There exists a simpler proof that goes through the CIU theorem:
By the CIU theorem,
STS: $\forall$ K $\div$ τ. K[callcc···] $\downarrow\downarrow$ K[if ···]

Assume the left-hand side terminates:
K[callcc ···] $\mapsto^*$ K[if e_0 then e_1[cont K/k] else e_2[cont K/k]].
We used the fact that $k \notin fv(e_0)$.

STS: K[if e_0 then e_1[cont K/k] else e_2[cont K/k]] $\downarrow\downarrow$
K[if e_0 then callcc(k.e_1) else callcc(k.e_2)].

The proof would be easy if $e_0 \downarrow v \in \{\text{true,false}\}$. But since

our language has callcc, it's not obvious that it must (despite the fact that the left-hand side terminates). How do you prove "CIU equivalence of evaluation contexts"?

In the LR proof, this is straightfoward. You start with $e_0 \approx e_0$ and show that the left-hand continuation is related to the right-hand contiuation.

Ie, the continuation • then e_1[cont K/k] else e_2[cont K/k] and the continuation • then callcc(k.e_1) else callcc(k.e_2) are equivalent.

Question (Deepak): We're reasoning about (true,true) or (false,false). Somehow we've "lost" the case that e_0 captures its continuation.

Answer: That's not actually what we're proving with ⊤⊤-closure. We never assume that the values we plug in on both sides are the result of evaluating e_0. All we need to show is that if we're passed related values, we do something good. We do not need to connect these related values to evaluation of e_0. There's nothing in such proofs to say that e_0 evaluates to true or false. (This is particularly true once you have state.)

This is related to the fact that it was very easy to add continuations to the model. The model does not equate as many terms as we would like. It only relates terms that are equivalent in the presence of callcc. It does not relate terms that are not equivalent in the presence of callcc.

In particular,

$\lambda f.\ f(0) + f(1) \equiv \lambda f.\ f(1) + f(0)$

in System F.

None of our ⊤⊤-closed models will let us prove this equivalence since we cannot prove $(0,1) \in V[\text{int}]$. We could prove it in a model that doesn't "observe evaluation order".

(Aside: Derek is pretty sure we can prove this in the step-indexed model coming next week.)

We know the semantics of System F is "context free". Evaluation doesn't care what the context is. In our setting with callcc, you can't just swap ordering. Either side may capture continuations. Here's a distinguishing context:

callcc(k.•(λx.throw x to k))

The point: k can do different things based on x=0 or x=1. We're taking advantage of one power of callcc: The ability to just "return" from a function. We've thrown goto into the language: Whenever you communicate with an uknown piece of code, you have no idea what will happen next.

Aside from Scott/Derek: In a sense, callcc is an effect.

— General recursive types

We've already seen one motivating example for step-indexing. The original motivation was to model recursive types in general (ie, without the strictly positive restriction).

We want

‡ $V[\mu\alpha.\tau]\rho := \{ (\text{fold } v_1, \text{fold } v_2) \mid (v_1,v_2) \in V[\tau[\mu\alpha.\tau/\alpha]]\rho \}$

⟺

$V[\mu\alpha.\tau]\rho := \{ (\text{fold } v_1, \text{fold } v_2) \mid (v_1,v_2) \in V[\tau]\rho,\alpha\mapsto\mu\alpha.\tau \}$

The problem last Tuesday was the type gets bigger, in general, on the right hand side. We saw that while this equation was true (assuming positivity), we couldn't define it this way. If α appears negatively in τ, we'd be taking a fixed point of a non-monotone function.

(Aside: Derek thinks if you can assume that such an equation exists for arbitrary τ, then you can derive a contradiction. Consider

$\mu\alpha.\alpha\to\alpha$

This is the type of the domain of arrows in the untyped lambda calculus:

app : $D \to (D \to D)$ "= unfold"

lam : $(D \to D) \to D$ "= fold"

Derek will get back to this example on Thursday after some thought.)

We saw ‡ isn't even a well-founded definition. The idea of step-indexing is to force a related definition to be well-founded.

We'll first do a slight variant of the model from

> Ahmed, 2006: “Step-indexed syntactic logical relations for recursive and quantified types”.

It's a nice paper. This new version fixes bugs in the ESOP'06 version.

> (Aside: Amal goes way below the level of pedantry Derek expects. This TR is around 200 pages.)

The flaw in the ESOP paper: She was arguing that her LR was complete wrt contextual equivalence. She used a similar argument to Pitts'. She relied on this equivalence-respecting property we saw. The problem: In the case of one type constructor, the logical relation is not equivalence-respecting in the presence of existentials, one of her type constructors. We'll get to this when we get to it. (The solution is very simple if you want a complete relation: Add $\top\top$-closure or remove existentials.)

Scott: You can encode existentials as universals. So why is there a problem with existentials but not universals?

> Answer: Recall how we extended the model with existentials:
>
> (a) $V[\exists\alpha.\tau]\rho := \{ (\text{pack } [\sigma_1,v_1] \text{ as } \exists\alpha.\tau, \text{pack } [\sigma_2,v_2] \text{ as } \exists\alpha.\tau) \mid \exists R \in \text{Cand}.\ (v_1,v_2) \in V[\tau]\rho,\alpha\mapsto R \}$.
>
> Recall the Church encoding for existentials:
> $\exists\alpha.\tau := \forall\beta.(\forall\alpha.(\tau\longrightarrow\beta)) \longrightarrow \beta$.
>
> Think of this as a kind of double-negation. It's a kind of $\neg\neg(\exists\alpha.\tau)$. For all continuations that expect some existential package, you can apply them and get some good result out. You can instantiate those continuations in some way that makes them happy; that's the double negation point.
>
> By the principle of representation independence, if two things are related by $V[\exists\alpha.\tau]$, then their η-expansion will be related.
>
> The other direction doesn't work.
>
> If two existential packages are contextually equivalent, that's a lot like saying they're related by the double

negation. For any continuations you're given, you can instantatite them in related ways. But our Church encoding doesn't force you to use its continuation. Put another way, just because you have related existentiaal packages, does not mean the underlying values are related candidates.

(b) $V[\forall\beta.(\forall\alpha.(\tau\longrightarrow\beta))\longrightarrow\beta.] =$"kinda" $\{ (p_1,p_2) \mid \forall(k_1,k_2)\in[\forall\alpha.\tau\longrightarrow\beta].\ (k_1\ p_1,k_2\ p_2)\in E[\beta] \}$

We have (a) $\subseteq$ (b) but not (b) $\subseteq$ (a). (b) doesn't say how k_1 p_1 and k_2 and p_2 are related but (a) does.

— Step-indexed model

The basic idea is surprisingly simpleminded. It was deemed a hack initially. With use, it's earned respect. It's a very good idea:

> Index all the relations by a natural number n representing the number of steps of computation for which the things "act equivalently".

That's the intuition. It doesn't hold up to scrutiny, but it's the rough idea.

In particular, if you want to know whether or not fold v_1 and fold v_2 "act equivalently" for n steps of computation, you have to know that v_1 and v_2 "act equivalently" for n-1 steps.

We will define

$$V[\mu\alpha.\tau]\rho := \{ (n,\text{fold } v_1,\text{fold } v_2) \mid \forall j<n.\ (j,v_1,v_2)\in V[\tau[\mu\alpha.\tau/\alpha]]\rho \}$$

> Aside: We could have used n-1 rather than j<n. Derek used this definition because Amal had.

There are different ways to write these things depending on how obvious you want it to be that the relations are well-founded. You only care about whether things are related for n steps. Here, that's defined in terms of relatedness for strictly fewer than n steps.

The whole construction will be defined first by induction on n and second by induction on τ.

Since we're writing this as a set rather than as a relation, this may look a little dodgy. If you view the RHS as a relation, it's clearly

well-defined:

$V\_m[\mu\alpha.\tau]\rho := \{ (n,\text{fold } v_1,\text{fold } v_2) \mid \forall n<m.\ \forall j<n.$
$(j,v_1,v_2) \in V\_n[\tau[\mu\alpha.\tau/\alpha]]\rho \}$

The point: When the step-indexing gets really tricky, you can make the notation heavier to show that the thing is well-defined.

Question (Deepak): Can we work with an n-indexed family of binary relations? What will break?

> Answer: It's hard to say exactly what will break. In ends up being equivalent in this setting. Ultimately, we care about the union of all these V_m's.

Question: When n is zero, what happens?

> Answer: Informally, there are no steps of computation to use, so there's nothing to show. Formally, the fuss with step indexing becomes vacuous; you need only prove the side-conditions (here, that the values are folds).

Our new definition of candidates bakes in a downard-closure condition with respect to the step indices:

$\text{Cand} = \{ R \in \text{Sub}(\mathbb{N} \times \text{CVal} \times \text{CVal}) \mid$
$\forall (n,v_1,v_2) \in R.\ \forall j<n.\ (j,v_1,v_2) \in R \}$

We only want to talk about relations that are downard-closed. This is an instance of a more general pattern we'll see when we get to (more general) Kripke models. You bake downward closure over the Kripke order into Cand. (Here, by analogy, j is the future world.)

$V[\text{bool}]\rho := \{ (n,v_1,v_2) \mid v_1 = v_2 = \text{true} \vee v_1 = v_2 = \text{false} \}$

> Aside: There are versions of the model where, at n=0, you relate all values. (So that truth at step 0 is trivial.)

$V[\sigma\times\tau]\rho := \{ (n,\langle v_1,v'_1\rangle,\langle v_2,v'_2\rangle) \mid$
$(n,v_1,v_2) \in V[\sigma]\rho \wedge (n,v'_1,v'_2) \in V[\tau]\rho \}$

> In this case: Because the type got smaller, the step-index could remain the same. It didn't have to. We could have replaced n's on the right by n-1's.
>
> We're building monotonicity into the relation. Because

monotonicity is preserved inductively by the type, we don't need to do anything in this case.

$V[\sigma\to\tau]\rho := \{ (n,\lambda x.e_1,\lambda x.e_2) \mid \forall j\le n.\ \forall v_1,v_2.$
$(j,v_1,v_2) \in V[\sigma]\rho \implies (j,e_1[v_1/x],e_2[v_2/x]) \in E[\tau]\rho \}$

Aside: We need only ordinary lambdas in the language.
Fix is a derived from once you have general recursive types.
(Aside from Dave: Derek explains the derived from on 18 Dec.
For the record:

fix f(x).e := λy.(unfold v) v y

where v := fold(λz.(λf.λx.e) (λy.(unfold z) z y))
and y, z ∉ fv(e).)

We're following Amal for the moment. (There are other ways to do this. We could have used (n,v_1,v_2) rather than forcing related values to be lambdas.)

The type gets smaller. We don't need strict j<n as we did with fold.

We quantify over all j's other than n so that the logical relation is downward-closed. We didn't need to do it in the case for pairs because we can prove inductively that we don't need to. The point: When we go to prove validity, we'll need j≤n at function types. We won't need it at pair types. This is very common with Kripke models: You bake downard closure into the definition at arrow types so your proof goes through.

$V[\forall\alpha.\tau]\rho := \{ (n,\Lambda\alpha.e_1,\Lambda\alpha.e_2) \mid \forall\sigma_1,\sigma_2 \in \text{CTyp}.\ \forall R \in \text{Cand}.$
$(n, e_1[\sigma_1/\alpha], e_2[\sigma_2/\alpha]) \in E[\tau](\rho,\alpha\mapsto R) \}$

Note we don't need j. Since there's no negative occurrence of the logical relation, we can push around n.

Etcetera.

Aside: There's a question that comes up. We're trying to determine whether two things are related at n steps. In ∀α.τ, we quantify over Cand (relating stuff at arbitrary steps). We only care about the piece of R that cares about n or fewer steps.

Next time, we'll define the relation on expressions and continuations.