— Stream example: Proof using admissibility

We can use admissibility to prove our streams example.

What Derek forgot last time: The admissibility condition is more general that what we tried to use.

Recall:

$\tau := \mu\alpha.1\to(\text{int} \times \alpha)$
ones : τ := fold (fix f ().<1,fold f>)
twos : τ := fold (fix f().<2,fold f>)

succ : $\tau \to \tau$:= fix f(s).
  let c = unfold (s()) in
  fold (λ().<1 + π_1c, f(π_2c)>)

Prop:

succ ones ≡ twos : τ.

We tried to set twos' := fold(λ().<1 + π_1<1,ones>, succ(π_2 <1,ones>)>) and to prove twos' ≈ twos using admissibility (Scott induction).

We got stuck in the case for n=1, trying to relate the non-recursive function "body of twos'" to the recursive function "body of twos".

Idea: It was the non-recursiveness of twos' that screwed us up.

The point: When you prove such equivalences using Scott indution, you need to work with recursive functions but you need not work with the same recursive function on both sides.

Recall Pitts' definition of the admissibility property:

∀n.(e[F_n/x],e'[F'_n/x]) ∈ E[τ]ρ
—
(e[F/x],e'[F'/x]) ∈ E[τ]ρ

In our proof attempt last time, we restricted ourselves to using this at e = e'.

Today, we'll try the proof with e = x and e' = succ. (Successor is a recursive function.)

Proof:

Set $e := x$
and $e' := \lambda().\langle 1+\pi_1\langle 1,\text{ones}\rangle, x(\pi_2\langle 1,\text{ones}\rangle)\rangle$.
With e', we've made a hole in twos' where succ had been.

Observe
> twos = fold (e[foo/x]) ∧
> twos' = fold (e'[succ/x]).

Set foo := fix f().<2,fold f>.

By the model and our unrolling of the μ last time,
STS: (e[foo/x],e'[succ/x]) ∈ E[1⟶(int × τ)]
⟸ (Admissibility)
> ∀n. (e[foo_n/x],e'[succ_n/x]) ∈ E[1⟶(int × τ)].

Proof by induction on n.

Case n = 0. Trivial. (Slightly less so than last time.)
> $e[\text{foo}_0/x] = \text{foo}_0$ and $\text{foo}_0()\uparrow$.
> $e'[\text{succ}_0/x] \mapsto^* K[\text{succ}_0\ \text{ones}]\uparrow$.

Case:
> Assume IH for every k<n.
> TS: (e[foo_n/x],e'[succ_n/x]) ∈ E[1⟶(int × τ)]
> ⟸(β)
> > (foo_n(),(e'[succ_n/x]) ()) ∈ E[int×τ].
>
> Observe foo_n() ↦* <2,fold(foo_{n-1})>
> and (e'[succ_n/x]) ()
> > ↦* <2,succ_n ones>
> > ↦* <2,fold (e'[succ_{n-1}]/x])>.
>
> STS: (fold(foo_{n-1}), fold (e'[succ_{n-1}]/x])) ∈ V[τ]
> ⟸(IH)
> > (foo_{n-1},e'[succ_{n-1}]/x]) ∈ E[1⟶(int×τ)].

Q.E.D.

The coninductive proof was nicer. But for this example, we didn't need the coinduction built into the model. In other words, for this particular example, we didn't care that the model was defined with gfp rather than lfp.

Aside: This proof justifies defining admissibility in terms of a pair of distinct terms e and e'.

— Adding callcc to the language

The natural next step is to move toward general recursive types. We'll not do that today.

We'll talk about another interesting feature—considered challenging to reason about—that becomes simple to reason about with ⊤⊤-closure: Continuations. This extension shows both the power and the limitations of ⊤⊤-closure. We'll return to this point later.

We'll follow Dreyer, Neis, and Birkedal (JFP, 2012) except we'll write cont(τ) as $\neg\tau$ for kicks.

Syntax:

Types	$\tau ::= \cdots \mid \neg\tau$
Terms	$e ::= \cdots \mid \text{callcc } (x.e) \mid \text{throw } e_1 \text{ to } e_2 \mid \text{cont}(K)$
Values	$v ::= \cdots \mid \text{cont}(K)$
Evaluation Contexts	$K ::= \cdots \mid \text{throw } K \text{ to } e \mid \text{throw } v \text{ to } K$

Idea: Callcc binds x to the current continuation (injected into a value form) and runs e. Cont(K) is the value form. Throw “applies” one of these values.

Statics:

$\Delta; \Gamma, x{:}\neg\tau \vdash e : \tau$
—
$\Delta; \Gamma \vdash \text{callcc } (x.e) : \tau$

Intuition: Logically speaking, this rule says $(\neg A \longrightarrow A) \longrightarrow A$. True classically, not intuitionistically. Idea: Continuations correspond to classical logic.

$\Delta; \Gamma \vdash e_1 : \tau$
$\Delta; \Gamma \vdash e_2 : \neg\tau$
—
$\Delta; \Gamma \vdash \text{throw } e_1 \text{ to } e_2 : \sigma$

Intuition: From $\tau \wedge \neg\tau$, you can conclude anything.

Aside: If you want the language to have unique types, you can annotate callcc and throw with types. (We've not bothered thus far.) This'd be equivalent to concluding with $\forall\alpha.\alpha$ rather than σ except it wouldn't mix concepts.

$\Delta; \Gamma \vdash K \div \tau$
—
$\Delta; \Gamma \vdash \text{cont } K : \neg\tau$

Aside: In the paper we're following, they defined typing for full contexts C. We've already defined $\Delta; \Gamma \vdash K \div \tau$ using the specialization of those C rules to evaluation contexts K.

Dynamics:

$K[\text{callcc}(x.e)] \longmapsto K[e[\text{cont}(K)/x]]$

$K[\text{throw } v \text{ to } (\text{cont } K')] \longmapsto K'[v]$

Note: We've ensured typing preservation, not type preservation. The whole program has a type on either side of $\longmapsto$. But the types differ.

Note: We use β reductions locally during proofs involving the model. With this reduction rule, we cannot reduce whole programs. (In our proofs, we've only ever used closure under expansion using β for types like $\longrightarrow$ that work in arbitrary contexts.)

— Adding callcc to the ($\top\top$-closed) model

NB this is a completely orthogonal extension. We don't have to change E[] and K[].

Our existing proof of closure under expansion only used β-reductions other than this one. This reduction obviously won't fit into that lemma. But we can restrict the statement of the lemma to those notions of β for which it works.

We have to extend the model. The blindingly obvious thing works:

$V[\neg\tau]\rho := \{ (\text{cont } K_1, \text{cont } K_2) \mid (K_1, K_2) \in K[\tau]\rho \}$

All the trivial metatheory lemmas (ought to) go through. We'll stop to prove any that turn up in the compatibility lemma.

— Metatheory

Lemma (Compatibility for callcc):

$\Delta; \Gamma, x : \neg\tau \vdash e_1 \approx e_2 : \tau$

—

$\Delta; \Gamma \vdash \text{callcc}(x.e_1) \approx \text{callcc}(x.e_2) : \tau$

Proof:

Let $\rho \in D[\Delta]$, $(\gamma_1,\gamma_2) \in G[\Gamma]\rho$, $\delta_1,\delta_2 \in \Delta \longrightarrow \text{CTyp}$.

TS: $(\delta_1\gamma_1\text{callcc}(x.e_1), \delta_2\gamma_2\text{callcc}(x.e_2)) \in E[\tau]\rho$

$\Leftarrow$ $(\text{callcc}(x.\delta_1\gamma_1e_1), \text{callcc}(x.\delta_2\gamma_2e_2)) \in E[\tau]\rho$.

Let $(K_1,K_2) \in K[\tau]\rho$.

TS: $K_1[\text{callcc}(x.\delta_1\gamma_1e_1)]\downarrow\downarrow K_2[\text{callcc}(x.\delta_2\gamma_2e_2)]$.

Since K_i[callcc(x.δ_iγ_ie_i)] ⟼ K_i[δ_iγ_ie_i[cont K_i/x]],

STS: $K_1[\delta_1\gamma_1e_1[\text{cont } K_1/x]]\downarrow\downarrow K_2[\delta_2\gamma_2e_2[\text{cont } K_2/x]]$.

Set γ'_i := γ_i,x⟼cont K_i.

WK: $(K_1,K_2) \in K[\tau]\rho$

$\Rightarrow$ $(\text{cont } K_1,\text{cont } K_2) \in V[\neg\tau]\rho$

$\Rightarrow$ $(\gamma'_1,\gamma'_2) \in G[\Gamma,x{:}\neg\tau]\rho$.

By premise,

$(\delta_1\gamma_1e_1[\text{cont } K_1/x], \delta_2\gamma_2e_2[\text{cont } K_2/x]) = (\delta_1\gamma'_1e_1,\delta_2\gamma'_2e_2) \in E[\tau]\rho$

$\Rightarrow$(Definition $E[\tau]\rho$)

$K_1[\delta_1\gamma_1e_1[\text{cont } K_1/x]]\downarrow\downarrow K_2[\delta_2\gamma_2e_2[\text{cont } K_2/x]]$.

Q.E.D.

Lemma (Compatibility for throw):

$\Delta; \Gamma \vdash e_1 \approx e'_1 : \tau$

$\Delta; \Gamma \vdash e_2 \approx e'_2 : \neg\tau$

—

$\Delta; \Gamma \vdash \text{throw } e_1 \text{ to } e'_1 \approx \text{throw } e_2 \text{ to } e'_2 : \sigma$

Proof:

By the bind lemma, this reduces to showing:

If $(v_1,v_2) \in V[\tau]\rho$

and $(v'_1,v'_2) \in V[\neg\tau]\rho$,

then $(\text{throw } v_1 \text{ to } v'_1, \text{throw } v_2 \text{ to } v'_2) \in E[\sigma]\rho$.

Let $(K_1,K_2) \in K[\sigma]\rho$.

TS: $K_1[\text{throw } v_1 \text{ to } v'_1]\downarrow\downarrow K_2[\text{throw } v_2 \text{ to } v'_2]$.

Unrolling $(v'_1,v'_2) \in V[\neg\tau]\rho$, there exist $(K'_1,K'_2) \in K[\tau]\rho$ satisfying

$v'\_i = \text{cont } K'\_i \quad (i \in \{1,2\})$.

By the dynamic semantics of throw,

STS: $K'_1[v_1]\downarrow\downarrow K'_2[v_2]$

$\Leftarrow$ $(K'_1,K'_2) \in K[\tau]\rho \wedge$

$(v_1,v_2) \in V[\tau]\rho$.

Q.E.D.

It's possible in this langauge to show a compatibility lemma for cont(K). It's not really necessary since we don't want programmers to write these things in their proofs. Avoiding it means our notion of contextual equivalence doesn't cover that type. We could prove it. For example, in the paper, Derek et al defined

$$\Delta; \Gamma \vdash K_1 \approx K_2 \div \tau := \forall\rho\in D[\Delta], (\gamma_1,\gamma_2)\in G[\Gamma]\rho, \delta_1,\delta_2 \in \Delta \longrightarrow \text{CTyp}.$$
$$(\delta_1\gamma_1K_1,\delta_2\gamma_2K_2) \in K[\tau]\rho$$

and then proved the compatibility lemma

$$\Delta; \Gamma \vdash K_1 \approx K_2 \div \tau$$
—
$$\Delta; \Gamma \vdash \text{cont } K_1 \approx \text{cont } K_2 : \neg\tau.$$

— A small example

This looks rather contrived, but Derek doesn't have too many examples of interesting proofs involving callcc. (We'll have better examples once we add state.)

This example is from Støvring and Lassen (POPL'07). It uses admissibility and callcc.

Define:

$\tau := \mu\alpha.(\neg\text{int})\longrightarrow\alpha$

$\varphi_1 := \text{fix } f(x{:}\tau){:}\text{int. callcc}(k.f((\text{unfold } x)k))$

$\varphi_2 := \lambda y{:}\tau.\text{callcc}(k.(\text{fix } f(x{:}\tau){:}\text{int. } f((\text{unfold } x)k))\ y)$.

Prop: $\varphi_1 \equiv \varphi_2 : \tau \longrightarrow \text{int}$.

Proof idea:

Idea: We have to reduce to the point where both sides have a k, then we'll have an admissibility proof.

Set F_k := fix $f(x{:}\tau){:}\mathrm{int}.\ f((\mathrm{unfold}\ x)(\mathrm{cont}\ k))$
so that $\varphi_2 = \lambda y{:}\tau.\mathrm{callcc}(k.F\_k\ y))$.

Let $(v_1,v_2)\in V[\tau]$, $(K_1,K_2)\in K[\mathrm{int}]$.
TS: $K_1[\varphi_1\ v_1]\downarrow\downarrow K_2[\varphi_2\ v_2]$
$\Leftarrow(\beta)$
$K_1[\varphi_1\ ((\mathrm{unfold}\ v_1)\ (\mathrm{cont}\ K_1))] \downarrow\downarrow K_2[F\_\{K_2\}\ ((\mathrm{unfold}\ v_2)\ (\mathrm{cont}\ K_2))]$.

We'll pick this up next time.

— Pros and Cons

Point about callcc: It was easy to add continuations to our language, but continuations change contextual equivalence. In giving the context additional distinguishing power, we lose equivalences. Contexts with callcc can make more distinctions than contexts without.

Point about $\top\top$-closure: With $\top\top$-closure, it was very easy to add continuations to our model. That seems very nice, but it has a down-side: $\top\top$-closed models cannot prove some contextual equivalences.

The following example touches both points.

Example:

$\tau := (\mathrm{int} \to \mathrm{int})\to\mathrm{int}$
$e_1 := \lambda f.f(0)+f(1)$
$e_2 := \lambda f.f(1)+f(0)$.

Then $e_1 \equiv e_2 : \tau$ in the language without callcc but not in the language with callcc. (Exercise: Find a distinguishing context in the presence of callcc.) Moreover, even without callcc, you cannot prove the equivalence using our $\top\top$-closed model. Such proofs boil down to showing
$K_1[f(0)] \downarrow\downarrow K_2[f(1)]$
(e.g., via the bind lemma) while you want to have to prove
$K_1[f(0)]\downarrow\downarrow K_2[f(0)]$.