Here's some history on the callback with lock example.

Leading up to the POPL'09 paper, Derek had been looking for a more exicting example. He found a similar example in a paper on proving representation indepedence for OO programs. (It's a rather more complicated approach.) He tried to prove it using the POPL'09 model. Being the first thing they had thought of, that model is more complicated than the one they settled on for the JFP version of the paper. The proof was ad hoc and relied heavily on abusing the step-indices in the model to encode "windows of time" during which things were true. It turns out the example should not have been provable. It does not hold in the presence of higher-order state and callcc together.

In the original paper, they hadn't dealt with callcc. They did not have $\top\top$-closure. They were able to prove the example in that model only because they built the model wrong, basically.

At the same time, there are other examples that only hold in the absence of callcc. Today we'll talk about the ICFP'10 paper. We'll deal with a simpler example that the story so far cannot account for.

Jacob Thamsborg, one of Lars Birkedal's students, created the following example and showed it to Derek. It has two names. The Very Awkward Example and the Well-Bracketed State-Change Example.

Example (Very Awkward):

$\tau = (1 \longrightarrow 1) \longrightarrow \text{int}$

$v_1 = \lambda f.(f(); f(); 1)$

$e_2 = \text{let } x = \text{ref } 0 \text{ in}$

$\quad \lambda f.(x := 0; f(); x := 1; f(); !x)$

With just exceptions, these are inequivalent. Our f can, the second time it's called, invokes the whole mess again with a second f' that, as soon as its called, throws an exn to f which terminates with $x \hookrightarrow 0$. (You can find the distinguishing context in the paper.)

Suppose we have no callcc with higher-order state and no exceptions. Are these, intuitively, equivalent? This example boils down to what are called well-brackeded computations. Intuitively, when you call f the second time, it may cause x to go temporarily to zero but it will go back to one eventually. Every set to zero is followed by a set to one.

It's a nice example because such well-bracketed reasoning is not captured by the transition systems we've been using. One way to understand this example. There are two roles in the transition system. We have control over the data structure. We can make the transition from one back to zero. That should be OK. But when we call this function f—coming to us from the context—should not be able to move from one to zero. Internally, sure, but its externally observable behavior is that f stays where we leave it. The mechanism to handle this example is to distinguish between transitions that can be made internally to a computation and all transitions: Private vs public transitions. Public transitions are a subset of private transitions.

Our STS looks like

$$x \hookrightarrow 0 \longrightarrow \quad x \hookrightarrow 1$$

with a private transition going the other way and the necessary (public) self transitions. (Derek draws private transitions with dashed arrows.)

We have to prove that the computation

x := 0; f(); x := 1; f(); !x

makes a public transition. Internally, it can make private moves. We limit the callback f—the context—to making only public transitions. This kind of reasoning has a certain rely-guarantee flavor. You show that your computation makes a public move but you may assume that other computations that may interfere with you make only public moves.

As another example, here's the transition system for the callback with lock example. Recall:

$\tau = ((1 \longrightarrow 1) \longrightarrow 1) \times (1 \longrightarrow \text{int})$
$e_1 = C[f(); x := !x + 1]$
$e_2 = C[\text{let } n = !x \text{ in } f(); x := n + 1]$
C = let b = ref true in let x = ref 0 in
<λf.if !b then (b := false; •; b := true) else (),
λ_.!x>.

The transition system has

• Countably many Unlocked States with public transitions between them:

$(b \hookrightarrow \text{true}, x \hookrightarrow 0) \longrightarrow$
$(b \hookrightarrow \text{true}, x \hookrightarrow 1) \longrightarrow$

(b↪true,x↪2) ⟶
...

• Countably many locked states:

(b↪false,x↪0)
(b↪false,x↪1)
...

• Private (or public) transitions from (b↪true,x↪n) to (b↪false,x↪n) for every $n \in \mathbb{N}$.

• Private transitions from (b↪false,x↪n) to (b↪true,x↪n+1) for every $n \in \mathbb{N}$.

— HW

For HW: Come up with a variant of the callback with lock example that requires the full transition system and cannot be proven with the simpler STS

(b↪true,x↪_) (b↪false,x↪n)

with two private transitions: One labelled n going from left to right and one unlablled going from right to left.

—

We'll now formalize these private/public transitions.
[Derek threw up the model in his JFP paper.]

• In an island, we now have two STS rather than one. The public one is a subset of the private one.

• We use the private STS in $V[\tau]\rho$, quantifying over arbitrary future worlds.

• We use the public STS in $K[\tau]\rho$, quantifying over public future worlds.

It makes sense for the continuation relation to be monotone wrt public future worlds. (The best way to see this is to work out, say, the proof of Very Awkward.)

Why don't these example work with callcc? The problem is very simple. Values have a stronger monotonicity condition than continuations. With

callcc, you throw continuations into the value language. You can take any continuation and view it as a value that can be passed around. This forces you to merge public and private transitions: Any monotonicity property that must hold for values must hold for continuations as well, once continuations become first-class.

— Other things in the paper

Inconsistent states and their motivating example: We may return to this later. The thing that's interesting about this example is there are calls to some f in related places. The functions you're passing f are clearly not equal. The example is tied to a particular property of divergence. In an inconsistent state, the two programs are inconsistent. One has terminated and the other has diverged. If you've reached such a state, you know you'll be able to prove a contradiction later in the proof. (The program on the left will not terminate, but you've assumed it will terminate.) See the paper for details.

Backtracking. This permits you to deal with, say, the callback with lock example in the presence of first-order state and continuations. We have to do away with the private transitions, but we can use backtracking to behave as if we still had them. (The problem isn't first-order vs higher-order state. The problem: The HOS model uses world-indexed heap relations.)

Exceptions. Unlike continuations, exns do not prevent you from using private transitions. Exceptions cannot just return to your earlier scope. See p.30 in the paper.