For the rest of the course, we'll use logical relations to reason about mutable state. Many of the techniques we've developed so far will scale up naturally. We'll also need so-called Kripke logical relations. The idea is to index the logical relation by a possible world used to encode invariants on local state. /Local state/ is some piece of mutable state that's kept private to a module (so that the module can impose some invariants on it).

There are two dimensions that lead to different, interesting approaches.

• What kind of state are we talking about?

We'll start today with simple, first-order state. That means refs only to values of base type. For simplicity, we'll use refs to integers.

Higher-order state means you can have references to values of arbitrary types. It's more complicated to model for reasons that are similar to the reason that general recursive types were more difficult to model than restricted recursive types: Naive attempts hit circularity. (The circularity comes up in the definition of possible worlds, rather than directly in the definition of the logical relation.)

Aside: First-order vs higher-order does not just determine how you technically set up the model. You end up with different in reasoning principles.

• What kind of invariants? We'll start model that allows you to establish a fixed invariant about local state (Pitts and Stark, 1998). We'll get to so-called transitional invariants (Ahmed-Dreyer-Rossberg, 2009). You allow the invariants over state to change over time in a structured way.

These dimensions are orthogonal. While ADR'09 addressed both higher-order state and transitional invariants, we'll cover them one at a time. The structure of the next several lectures:

1. First-order state with simple invariants.
2. Transitional invariants.
3. Higher-order state.
4. Even richer forms of transitional invariants.

(Probably based on Dreyer, Neis, Birkedal 2010).

The invariants we'll cover in (4), unlike those in (2), sometimes come with restrictions; for example, some only work in the absense of continuations.

— Mutable state

Let's start by extending the language we've worked with so far with first-order mutable state.

Syntax:

$\tau ::= \cdots \mid \mathrm{ref}$
$e ::= \cdots \mid \mathrm{ref}\ e \mid !e \mid e_1 := e_2 \mid \mathrm{ell}$
$v ::= \cdots \mid \mathrm{ell}$
$K ::= \cdots \mid \mathrm{ref}\ K \mid !K \mid K := e \mid v := K$

We've a base type: References to integers, the obvious ML-style expressions, and pointer values as expressions. We assume base types int and 1 (with suitable constants). We've seen these before.

Statics:

$$\frac{\Delta; \Gamma \vdash e : \mathrm{int}}{\Delta; \Gamma \vdash \mathrm{ref}\ e : \mathrm{ref}}$$

$$\frac{\Delta; \Gamma \vdash e : \mathrm{ref}}{\Delta; \Gamma \vdash !e : \mathrm{int}}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathrm{ref} \quad \Delta; \Gamma \vdash e_2 : \mathrm{int}}{\Delta; \Gamma \vdash (e_1 := e_2) : 1}$$

What about ell? A typical way to prove soundness is to extend the typing judgements with

$\Sigma ::= \emptyset \mid \Sigma, \mathrm{ell}$

and track allocated locations. These Σ's just get propogated through, without interacting with everything, except we might add a rule

$$\frac{}{\mathrm{ell} \in \Sigma}$$

$\Sigma; \Delta; \Gamma \vdash \text{ell} : \text{ref}$

The point of such a rule: We only refer to ell's that have been allocated. It's useful when you try to prove preservation.

We're not going to take this approach for the simple reason that we're going to build an untyped model. So instead, we have no typing rule for locations.

This means ell's may only come up dynamically. You cannot write a well-typed program with free locations.

Dynamics:

> We change the judgements we had before, adding heaps (and tweaking the arrows so things remain readable going forward).
>
> Heaps $h ::= \emptyset \mid h, \text{ell} \mapsto v$
>
> Judgement $h; e \hookrightarrow h'; e'$
>
> > All of the rules from before carry over to the new judgement.
>
> $\text{ell} \notin \text{dom}(h)$
> —
> $h; K[\text{ref } v] \hookrightarrow h,\text{ell} \mapsto v; K[\text{ell}]$
>
> —
> $h,\text{ell} \mapsto v; K[!\text{ell}] \hookrightarrow h,\text{ell} \mapsto v; K[v]$
>
> —
> $h,\text{ell} \mapsto \_; K[\text{ell} := v] \hookrightarrow h,\text{ell} \mapsto v; K[()]$

— An example: Reasoning about local state

Aside: Pitts and Stark offer several more interesting examples. We'll get to even more interesting examples when we discuss more recent papers.

This example is somewhat like an example we saw in the pure setting when we worked with existentials. (Aside: Generally, we can hide state

using type abstraction or, imperatively, with local state. More on this point later.)

Think of a counter object with two methods: Increment the counter and get its current value.

$\tau = (1 \longrightarrow 1) \times (1 \longrightarrow \text{int})$
$e_1 = \text{let x = ref 0 in } (\lambda\_.x := !x + 1, \lambda\_.!x)$
$e_2 = \text{let x = ref 0 in } (\lambda\_.x := !x - 1, \lambda\_.0 - !x).$

Aside: We can encode let:
$\text{let x} = e_1 \text{ in } e_2 = (\lambda x.e_1)e_2.$

We'd like to prove $e_1 \equiv e_2 : \tau$. Intuitively, it makes sense. We can impose a relational invariant on the local pieces of the heap owned by the e_i:
Whenever x is n for e_1, then it's -n for e_2.

Fuzzy foreshadowing:
Pretend we've run these programs, allocating ell_1 for x in e_1 and ell_2 for x in e_2. Then the island we want looks like
$\{ ([\text{ell}_1 \mapsto n],[\text{ell}_2 \mapsto -n]) \mid n \in \mathbb{Z} \}.$
where the notation $[\text{ell}_1 \mapsto n_1, \cdots, \text{ell\_k} \mapsto \text{n\_k}]$ means the heap $\emptyset, \text{ell}_1 \mapsto n_1, \cdots, \text{ell\_k} \mapsto \text{n\_k}$.

Aside from Dave: Such islands can represent “one-sided” heap invariants; for example,
$\{ (\emptyset, [\text{ell} \mapsto n]) \mid n \in \mathbb{N} \}$
relates the empty heap on the left to all heaps h satisfying dom(h) = { ell } and h(ell) ≥ 0 on the right.

Here, the thing that's abstract is the invariant on the local state. It doesn't show up in the types at all. (The same kind of reasoning that showed up with our existential examples will come up.)

The kinds of reasoning you can do with local state is much richer than the kinds of reasoning you can do with abstract types. There's a lot of interesting stuff going on in the semantics of state that isn't reflected in the simple type system for existentials. The type system has fixed, abstract types. For each abstract type, you pick a fixed relation. That gives you a fixed invariant. Particularly interesting (and the motivation for the POPL'09 paper) is when you combine invariants on local state with invariants on abstract types.

— Kripke logical relations

The idea is to formalize the notion of “invariants on local state” using a “possible world” W. A possible world encodes all of the invariants on the local state of the terms that we're reasoning about.

We'll end up with something of the form

$$(W,e_1,e_2) \in E[\tau]\rho$$

Meaning e_1 and e_2 are related under world W. Unlike the past, we no longer know the e_1 and e_2 behave the same under any possible assumptions. What we know is that under invariants about heaps (that show up as e_1 and e_2 run) encoded in W, e_1 and e_2 behave equivalently.

Diverging from Pitts and Stark, we'll retain our step indices. Moreover, a world W will be a tuple of “islands” where an island is just a relation on heaps govering its own piece of the “whole” heap. (What Pitts and Stark do amounts to having a single island.)

Some useful sets:

$$\text{Island} = \text{Sub}(\text{Heap} \times \text{Heap})$$

(An island $\iota \in \text{Island}$ is a relation on heaps.)

$$\text{World} = \{ W = (j,\omega) \mid j \in \mathbb{N} \wedge \exists n.\ \omega \in \text{Island}^n \}$$

(A world $W \in \text{World}$ comprises a step index and a tuple of islands.)

Where Island^n is just an n-tuple of islands. Each island in the tuple will provide invariants for a different “module” or piece of local state. In our proofs, we'll start with some unkown world W and, as needed, extend the world with a new island governing “our” local state. It'll be like width extension in record subtyping. That this lets you hold on to your invariants will become clear when we define the future world relation.

World extension: We define $W' \sqsupseteq W$ (read W' extends W).

$$W' \sqsupseteq W \text{ if } W' = (j',\omega') \wedge W = (j,w) \wedge j' \leq j \wedge \omega' \in \text{Island}^{\{n'\}} \wedge \omega \in \text{Island}^{\{n\}} \wedge n' \geq n \wedge \forall i \in 1..n.\ \omega'.i = \omega.i$$

Think of W' as the future and W as the past. We used ω.i to denote the

ith island in the tuple ω. The definition ensures that the step-indices make sense and ω is a prefix of ω'.

Future world operator: We also define a future world operator that we'll only ever apply when we know the current world isn't at step-index zero:

$$\triangleright W = (j\text{-}1,\omega) \text{ if } W = (j,w) \wedge j>0$$

Note $\triangleright W$ is undefined when $W.j = 0$.

Aside: Pitts and Stark do not use such syntactic islands. They use a separating conjunction to achieve the same effect. In more expressive models, their approach may be more flexible than what we're doing, but Derek hasn't had that come up to date.

World satisfaction: We use these worlds to impose invariants. We want to know when two heaps satisfy these invariants. Read $h_1,h_2 : W$ as "h_1 and h_2 satisfy W".

$h_1,h_2 : W$ if
$W = (j,\omega) \wedge \omega \in \text{Island}^n \wedge$
$\forall i \in \{1,2\}. \exists h^1\_i, \cdots, h^n\_i.$
$(h\_i = h^1\_i \uplus \cdots \uplus h^n\_i \wedge$
$\forall k \in 1..n. (h_1\hat{}k, h_2\hat{}k) \in \omega.k).$

Monotonicity: Our earlier notion of montonicity wrt downward closure generalizes to world extension. A key property we'll ensure is montonicity of the value relation with respect to world extension. (Aside: We'll continue to use this phrase world extension later on when it stops being, in some sense, the right phrase.)

Monotonicity:
If $(W,v_1,v_2) \in V[\tau]\rho$,
then $\forall W' \sqsupseteq W. (W',v_1,v_2) \in V[\tau]\rho$.

Intuitively, we care about monotonicty so that we can add "other" invariants on disjoint pieces of state without ruining our "current" invariants. If I claim that two values are equivalent, I may give them to you without knowing when you'll use them. My invariants have to survive even if you impose your own invariants on your own local state.

The value relation at function types:

$$V[\sigma\longrightarrow\tau]\rho = \{\ (W,\lambda x.e_1,\lambda x.e_2) \mid \forall W' \sqsupseteq W.$$
$$(W',v_1,v_2) \in V[\sigma]\rho \Longrightarrow$$
$$(W',e_1[v_1/x],e_2[v_2/x]) \in E[\tau]\rho\ \}$$

The proof of congruence of the logical relation will ensure that there's nothing some well-typed piece of code can do to ruin some other module's local invariants.

The value relation at ref types:

$$V[\text{ref}]\rho = \{\ (W,\ell_1,\ell_2) \mid \exists i.\ W.\omega.i =$$
$$\{\ ([\ell_1 \mapsto n],[\ell_2 \mapsto n]) \mid n \in \mathbb{Z}\ \}\ \}.$$

Idea: Related references store the same value.

Aside from Dave: Islands impose invariants on heaps. Actual heaps come into play via world satisfaction. So V[ref] says W has an island imposing an invariant that forces any heaps $h_1,h_2 : W$ to satisfy $h_1(\ell_1) = h_2(\ell_2)$.

Aside: One can use semantic models like ours to prove safety of the language. (Often you don't need such fancy models for safety, but sometimes you do.) With this kind of language, if we only wanted to prove safety, we could restrict our islands to be of a specific form, comprising the invariant we stick at the ref type: Related references store the same value. That's the least we must do.

Aside: The language permits us to impose more invariants using our islands. Well-typed programs are guaranteed not to care about invariants other than those on ref types.

Two locations are related when the world enforces the invariant that they're related. We're not saying anything about what their contents are. We don't know their contents. Their contents will show up in some pair of heaps h_1 and h_2 satisfying W. We'll know, by the invariant, that the contents of ℓ_1 and ℓ_2 in h_1 and h_2 are the same integer.

The invariant we impose in V[ref] are precisely those we need to prove compatibility at ref type.

There are many different ways of handling this type.

Question: Why focus on singleton heaps?

Answer: Good question. It depends on how much you want to prove about ref types. Ref types are actually quite poorly behaved. There are many very subtle equivalences involving ref types that are not robust under mild changes to the language. (Maybe not so much with this language, but certainly in the higher-order case.) There are disgusting equivalences that shouldn't hold but do because of an accident.

General references are a useful but somewhat unfortunate compromise (when it comes to actually doing verification). When you put ref types in the interface of (say) an ML module, the module's clients can do things to them. The module can no longer impose invariants on those values.

The point: There are different ways to model the ref type. How you model the ref type gives you different ways to reason about equivalences. You can impose more and more subtle invariants on the ref type that allow you to prove more equivalences. Derek has come to the conclusion that you should not worry about ML-style references as their properties with respect to contextual equivalence aren't so hot.

The value realtion at other types is pretty much what you expect.

The expression and continuation relations:

$$E[\tau]\rho := \{ (W,e_1,e_2) \mid \forall K_1, K_2. \\ (W,K_1,K_2) \in K[\tau]\rho \Longrightarrow \\ (W,K_1[e_1],K_2[e_2]) \in O \}$$

$$K[\tau]\rho := \{ (W,K_1,K_2) \mid \forall v_1,v_2.\ \forall W' \sqsupseteq W. \\ (W',v_1,v_2) \in V[\tau]\rho \Longrightarrow \\ (W',K_1[v_1],K_2[v_2]) \in O \}$$

where O is a set of observations (we finally talk about heaps and world satisfaction):

$$O = \{ (W,e_1,e_2) \mid \forall h_1,h_2 : W.\ h_1,e_1 \downarrow\downarrow W.j\ h_2; e_2 \}$$

where

$$h_1,e_1 \downarrow\downarrow W.j\ h_2; e_2 \text{ if} \\ (h_1;e_1\downarrow \wedge h_2;e_2\downarrow) \vee$$

$(h_1;e_1 \hookrightarrow W.j \wedge h_1;e_2 \hookrightarrow W.j)$

This is a slick presentation of the double implication “If one of them terminates in W.j steps, then the other one terminates (and vice-versa).”. If one of them gets stuck in W.j steps, then they won't be in O.

Open logical relation:

$D[\Delta] := \{ \rho \in \Delta \longrightarrow \text{Cand} \}$
$G[\Gamma]\rho := \{ (W,\gamma_1,\gamma_2) \mid \forall(x{:}\tau) \in \Gamma.\ (W,\gamma_1 x,\gamma_2 x) \in V[\tau]\rho \}$

$\Delta; \Gamma \vdash e_1 \approx e_2 : \tau$ if
$\Delta; \Gamma \vdash e_1 : \tau \wedge$
$\Delta; \Gamma \vdash e_2 : \tau \wedge$
$\forall \rho \in D[\Delta].\ \forall (W,\gamma_1,\gamma_2) \in G[\Gamma]\rho.\ \forall \delta_1,\delta_2 : \Delta \longrightarrow \text{CTyp}.$
$(W,\delta_1\gamma_1 e_1,\delta_2\gamma_2 e_2) \in E[\tau]\rho.$

Next time, we'll show some cases for soundness and move on to more interesting worlds.