— From the sacred (Chrome/Flash bugs) to the profane (Environmental Bisimulations).

The Sumii-Pierce POPL'05 work led, in part, to Derek's interest in logical relations. Derek recommends it. It introduces things well, even if the technique is a bit dated.

Back to Definition 6 in the 2009 paper. How do we justify the claim that the reasoning with these environmental bisimulations is in the same style as the reasoning in our KLRs? Think of X as a state space. Think of $(\Delta,R,s,s')\in X$ as defining a state. The only heaps related in this state are s and s'. In that state, R says what values are related at what types. (Every element of R comprises two values and a type.) What is the transition relation between the states? It's basically baked in as the subset relation on R. If you're in a state (Δ,R,s,s'), you can transition to any other state (Δ',R',s_1,s'_1) where $R' \supseteq R$. You can see this in the Evaluation relation (Defintiion 6, 1b). This gives you one way to define a big transition relation. The only place this story is written down is the Hur-Dreyer POPL'12 paper. (The whole idea of thinking about KLRs as working over STS' in worlds had not been developed.)

What are the limitations, compared to KLRs? You can only encode public transitions, since the encoding of transitions is in terms of R getting bigger. Thus, the approach is fairly limited in terms of the kinds of transition systems you can encode. It works perfectly fine for the examples from the POPL'09 paper (Derek coauthor) but falls down with examples that require private transitions or other features.

Aside: There was an interesting, intermediate piece of work on step-indexed bisimulations. Basically, served as a stepping-stone from Sumii-Pierce (limited in its ability to reason about higher-order functions) to the work on parametric bisimulations. Reference:

> Koustavas, Wand.
> Small bisimulations for reasoning about higher-order imperative programs.
> POPL'06.

The paper is worth reading. It has some good examples. It makes the work on parametric bisimulations look simple.

— Parametric Bisimulations

> Hur, Dreyer, Neis, Vafeiadis.

The Marriage of Bisimulations and Kripke Logical Relations. POPL'12.

One way to understand this work is the proof obligation for functions in bisimulation approaches (page 3, (5)):

If $(\tau' \longrightarrow \tau, v_1, v_2) \in L$,
then $\exists x, e_1, e_2.$ v_i=λx.e_i
and $\forall v'_1, v'_2.\ (\tau', v'_1, v'_2) \in [G] \Longrightarrow (\tau, e_1[v'_1/x], e_2[v'_2/x]) \in L$.

The question: What is this [G] from which you draw the arguments? In Sumii's 2009 work, the arguments came from $(\Delta,R)_*$, basically the context closure of the “current state”.
That approach is /very/ syntactic. It requires you know that the syntax in boht languages is the same so you can close over contexts.
What other approaches can there be?

— Normal-form Bisimulations

There's another approach. It's called normal-form bisimulations (or open bisimulations). The basic idea: Instead of trying to come up with a set of values representing the related arguments, you instead just pick a new variable x represening this unknown thing. Instead of substitutiong $[v'_1/x]$ and $[v'_2/x]$, we just start reasoning about open terms. When you actually try to use x (for example if it's a function that you apply), you'll get stuck.

Citation: Støvring and Lassen. POPL'07.
Better (The original paper): Lassen. Eager normal form bisimulations. 2005.

See §3 in Lassen'05. Look at (enf.4). To deal with λs, you assume they have the same variables and you relate the (open) bodies. In (enf.2), we have a call to a function x in evaluation position. The proof obligation is basically that E and E' are related continuations.

Instead of trying to actually model what related things are at a point where you're trying to quantify over some values you're given, you just pick a fresh name and keep going. This works fine in the case where the thing you're being given is a function. The only thing you can do with a function is call them. When you call a function, you have the obligation to pass related arguments and related continuations. So this seems like a nice idea. There were several attempts (eg, Støvring-Lassen, POPL'07) to extend the approach. It gets rather complicated. As far as Derek can see, it was never developed enough to deal with the kind of reasoning we can do with recent KLRs.

— Back to Parametric Bisimulations

The goal was to develop something that wasn't tied fundamentally to syntax, is simpler than work on bisimulations, and has the full power of recent KLRs.

Recall

> If $(\tau' \longrightarrow \tau, v_1, v_2) \in L$,
> then $\exists x, e_1, e_2.$ v_i=λx.e_i
> and $\forall v'_1, v'_2.\ (\tau', v'_1, v'_2) \in [G] \Longrightarrow (\tau, e_1[v'_1/x], e_2[v'_2/x]) \in L$.

The trick in picking [G]: Don't!

Refer to Derek's POPL'12 slides, the slide on "Definition of consistent(~L)". We write ~L for local knowledge about what things are equivalent. There are two other relations to sort out: The relation for values (~1) and the relation for terms (~2). You have to quantify over values that are related by some larger equivalence (so you know your ~L isn't simply wrong). Rather than pick a global equivalence ~G, make ~G a parameter of the whole model. You make some assumptions about ~G; for example ~G ⊇ ~L. But otherwise, ~G can relate anything. The idea: If ~G relates some junk (4,true)∈~G, then you'll get stuck but that's OK so long as the arguments and continuations are related (following work on open bisimulations). You're calling "functions" that are related by ~G: It's not your fault you got stuck, it's the environment's fault. (It would be our fault if we passed the environment 0 and 3 rather than related ints.) This motivates taking ~exp_G as ~2.

What is ~exp_G? It formalizes the idea that our terms should behave equivalently "locally". We now have three cases. Related terms both diverge, both converge, or both get stuck calling some functions (f_1,f_2) related by ~G. In this case, you pass control to the context, get back some values r_1,r_2, assume they're related, then continue. Recall that when we did proofs in our KLR models (eg, the Awkward example), we reasoned in exactly this way.

So what's going on? We're encoding what our logical relations proofs looked like. (Obviously the model looks different.) The structure of the proofs is reflected in the definition of the bisimulation.

So why is this good? Since you formalize exactly how the proofs are structured, it's easier to compose them transitively. This comes back to the main limitation of the work on ML-Assembly. If you want to reason about “vertical compositonality”, then you need to be able to transitively compose equivalence proofs. When you have a highly-constrained structure wherein you know that the code is eventually going to call some unknown function, then the only way to do the proof is to use the case where you get stuck calling some function related by ~G. With KLRs, that's the only way Derek knows how to handle such proofs but the model does not impose that proof structure. There may be a clever or brute-force proof. In this method, the *only* way to prove such things is by using the “stuck” case of the bisimulation. Since you constrain how the proof is structured, you know the structure of the proof follows—in some sense—the structure of the code. Because of that, it's possible to compose the proofs transitively.

The paper handles modelling worlds, abstract types, etc. It's much like what we've done with KLRs.

An example (Very Awkward): See §7.1 in the paper. To do a proof, you define a world W that has some state space S.

When you prove the example using KLRs, you sort of reason at the “instance” level. With this model, you reason at the “class” level. You have to construct your worlds so that you can have any number of copies. (Hence the Loc $\rightharpoonup$ XXX in the example.)

When you set up a proof, you specify the islands you're going to use in your proof and you restrict yourself to worlds that only have those islands. KLRs are, in a sense, more general than necessary. We define the world space to include any possible island. But LR proofs extend the world with a few kinds of islands (typically one), tailored to the proof. What about the context? It could extend the world with other kinds of islands. Well that's true, but if you're just trying to show the relatedness of your two programs wrt contextual equivalence, then you only need to handle the kinds of islands introduced by the compatibility lemmas. (Recall that except for the ref type, compatibility lemmas had no interaction with worlds/islands.) In this technique, we worry only about the islands we care about plus “the ref island”. Things were set up so you can prove if two equivalences were estabilished using two different worlds, then you can join things together.

This leads to a nice benefit: You can define your islands without using step-indexing.

Incidentally, this intuition came to Derek from looking at some papers on logics for storable locks (from the concurrency class a couple years ago). There's circularity in modelling such logics. There were different approaches to modelling these logics. One used step-indexing. The other approach was to state up-front “these are the kinds of assertions my proof cares about” and using that restricted view to deal with the circularity.

Back to the Very Awkward example.
The line

$w.L(s)(G)(\tau) := \{ (v_1,v_2[ell/x]) \mid ell\ \mu o\ dom(s) \}$

corresponds to stating in state s, I claim these things are related.

The interesting part of the proof: Now that you've defined this world, you have to show its consistent.

Limitations overcome since the model in POPL'12 (in a recent LICS submission).

• See the paragraph “The Trouble with η-Equivalence”. It requires introducing a notion of “stuttering” into the bisimulation so that you don't have to make progress immediately. You can not make progress on one side so long as you only do so finite many times. That's sufficient for η-expansion: Terms are finite so you only η-expand finitely many times.

• Continuations and control operators.

— Reasoning compositionally about open terms

Open equivalences are problematic. Type systems simply don't express enough information about free variables to be practically useful (for, say, verification in a single language).

Suppose we want to prove $A \approx B$. They both are clients of some C. What do I do? If C's type is too uninformative, it's not going to help; for example

$C{:}\tau \vdash A \approx B$

may not be provable. The equivalence may only be true because a specific C provides some specific functionality. But we still want to reason compositionally, without examining the implementation of C.

You could try

let C = CSpec in A ≈ let C = CSpec in B

where CSpec is some reference implementation of C; for example, a simple implementation of a hash table. You could then try to show

CSpec ≈ CActual

and use contextual equivalence to obtain the goal

let C = CActual in A ≈ let C = CSpec in B.

That's the kind of reasoning you'd expect to be able to do with contextual equivalence. Looks reasonable.

Two problems:

• You have to cough up the CSpec code. This is a basic limitation of the technique. You might argue that CSpec should be specified in some more abstract way (rather than by writing some code). Not a big deal.

• This approach assumes the code of the module that A depends on is "owned" by A. It's let-bound. What if we have

```
let C = CActual in
let Foo = Whatever in
let A = Whatever' in
...
```

where all these guys depend on CActual. So the hash table module is shared by all the clients, of which A is only one. Then the approach based on contextual equivalence falls down since it assumes A has complete control over the module. (Real programs work this way.)

So the main idea is to move away from contextual equivalence and move toward specifying the behavior you care about in some model/logic. Each client would have a private view. In some sense, this is what Neel's ICFP'12 paper (Superficially substructural types) was about.

Very interesting work: Merging this kind of reasoning with the kind of relational reasoning we've been doing in this course.