— History

Strachey (1967 lecture notes) introduces the term “parametric” vs “ad hoc” polymorphism. His canonical example of a parametric function is map : $\forall\alpha.\forall\beta.(\alpha \to \beta) \to \text{list}\ \alpha \to \text{list}\ \beta$.

(Aside: Derek is impressed that everyone cites Strachey for the one paragraph in his lecture notes devoted to this idea.)

Strachey gives us a very intentional notion of parametricy: A parametrically polymorphic function uses the same behavior (code) at different types.

We'll be studying an extensional notion of parametricity.

The first published notion of extensional parametricity: Girard's thesis.

Girard's thesis (1972) and Reynolds (1974) introduced System F. Girard cared about the proof theory of second order intuitionistic logic. He introduced “Girard's method” (aka, “Girard's method of candidates of reducibility”—see the cute paper Gallier (1992), introducing the method and arguing that most variations on it don't matter) for proving strong normalization of System F. That is, normalizations of well-typed programs terminate (for $\beta\eta$-reduction). This both limits the language's power (no nonterminating programs) and leads to very interesting reasoning principles. Leads to “unary” parametricity, the simplest model we might start with. (Aside: One way to think about a model of a language is as a very strong induction hypothesis. Compare to denotational semantics.)

Girard's basic idea: Abstract types represent sets that *do not* have to correspond to syntactic types in the language.

Example:

If $e : \forall\alpha.\alpha$
and $\text{fv}(e) = \emptyset$,
then contradiction.

With unary parametricity, we can *prove* that in System F, there are no terms of type $\forall\alpha.\alpha$. (Instantiate α with $\emptyset$. From the model, $e \in \emptyset$. Contradiction.)

Aside: System F vs ML. System F is impredicative: Type variables α can be instantiated with any type, even

polymorphic types.

Example:

If $f : \forall\alpha.\alpha \longrightarrow \alpha$,
then $f \approx \Lambda\alpha.\lambda x{:}\alpha.x$.

Using unary parametricity, we can prove this. (Boils down to applying f to a singleton set. Again, sets in the model do not have to correspond to syntactic types.)

Aside: Using set-based parametricity, we have proven two so-called "definability of types" results:

$\forall\alpha.\alpha \approx 0$ (void)
$\forall\alpha.\alpha \longrightarrow \alpha \approx 1$ (unit)

In fact, all such sums are definable in System F. We can't prove the next such result without relational parametricity.

$\forall\alpha.\alpha \longrightarrow \alpha \longrightarrow \alpha \approx 2$ (bool)

(Where true and false are the two inhabitants.)

Claim:

$\text{true} \equiv \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.x$
$\text{false} \equiv \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.y$
$\text{if b } e_1\ e_2 \approx \text{b } e_1\ e_2$

If $f : \forall\alpha.\alpha \longrightarrow \alpha \longrightarrow \alpha$,
then f is either true or false.

Failed proof: First, instantiate α to $\{v_1,v_2\}$. $(f\ (v_1)\ (v_2)) \in \{v_1,v_2\}$. So far, so good. But Maybe: f(3)(5) = 3 but f(3)(4) = 4. We need relational parametricity to prove that can't happen.

What we want to show is that with two calls f(3)(5), f(2)(4), we either get (3,2) or (5,4). We're relating two runs of the program.

Aside (odd claim by Derek, IMO):

Utility of n-ary method an open question. Odd since an n-ary relation is just a certain kind of unary relations. Not odd since unary vs binary parametricity are different models: In

> the unary case, Girard worked with sets of terms of the language. In the binary cases, the model changes.

Reynold's paper (1983): Types, abstraction, and parametric polymorphism. Intuition: Clients of an abstact type should be *invariant* under changes of representation of that type (ie, over multiple runs).

> Instead of instantiating α with a set of values, we instantiate it with a relation describing the change in representation of values. Eg, instantiate α with $\{(3,2), (5,4)\}$.

Aside: Another definability of types result is that one can encode existentials in System F (via CPS): $\exists\alpha.\tau \approx \forall\beta.(\forall\alpha.\tau \longrightarrow \beta) \longrightarrow \beta$.

Other applications of relational parametricity (beyond definability of types):

Free theorems (Wadler, 1989). For more interesting polymorphic types (than the identity), you know things that can be used in optimizations. For example, if $f : \forall\alpha.\text{list } \alpha \longrightarrow \text{list } \alpha$, then f can't inspect the list's elements. It might permute, drop, or duplicate elements. It's becoming clear that parametricity gives us properties that are hard to formally state. The free theorem for this type:

> $\forall g{:}\sigma \longrightarrow \tau.\ (\text{map } g) \circ (f[\sigma]) = f[\tau] \circ (\text{map } g)$.

This is an extensional property of f: We're beginning to characterize how f behaves. Such hoisting operations come up in optimizations. Short-cut fusion relies on such equational reasoning, enabled by parametricity. It's used in haskell implementations. It was proven sound in (Johann, 2002).

The applications so far are of the form What do you know about all things of a universal type? Derek's work: What do you know about some things of an existential type?

Canonical paper for “representation independence”: (Mitchell, 1986). ADTs and modules can be represented as terms of existential type ($\exists\alpha.\tau$). We can use relational parametricity to prove that two implementations of an ADT are contextually (or observationally) equivalent.

• It's good to know that modularity mechanisms in a particular language actually work. (Derek got started in this line of work because ML lacked a representation independence theorem.) The proof principle for existential types wind up being a kind of simulation argument.

• It's useful in showing that some implementation of an ADT matches the behavior of a reference implementation.

— Recent work

Earthly delights: Relational parametricity applied to langauges with recursion, recursive types, mutable state (tons of work on idealized algol, ML-style state studied “only” since the late 1990s), control operators, concurrency.

Basic starting point: Logical relations. Often considered introduced by (Tait, 1967), an inscrutable forerunner to Girard's thesis work. Girard uses Tait's method to prove strong normalization for the simply-typed λ-calculus. (Tait and Girard considered only unary logical relations.)

Basic idea: Build a model of the language defined by induction on the type structure.

Problems that have been encountered:

• Recursion: Logical relations are not automatically “admissible” (a domain theory term meaning they don't support fixed points). One thing you can do to make the construction admissible is restrict the relations involved to make things admissible. Another is biorthogonality (Pitts-Stark 1998): Account for the context in which terms are executed

• Recursive types: The problem with general recursive types. You try to define equivalence at type $\mu\alpha.\tau$ in terms of equivalence at that type. Induction on types no longer works. Idea: Do induction on something else. Step-indexing: Do induction on steps of computation. (Appel-McAllestor, 2001) and Ahmed's thesis (2005).

• Mutable state: Another form of data abstraction (distinct from and much more comman than abstract types): Local state. We want to reason about local state the same way we can reason about abstract types. Key idea: Kripke logical relations. A lot of work was done in the setting

of Idealized Algol. Parameterize the logical relation by a “possible world” encoding invariants on local state.