Lecture Notes on Parametricity

15-814: Types and Programming Languages
Frank Pfenning

Lecture 16
Tuesday, October 29, 2019

1 Introduction

> **Disclaimer:** The material in this lecture is a redux of presentations by Reynolds [Rey83], Wadler [Wad89], and Harper [Har16, Chapter 48]. The quoted "theorems" have not been checked against the details of our presentation of the inference rules and operational semantics.

As discussed in the previous lecture, parametric polymorphism is the idea that a function of type $\forall\alpha.\,\tau$ will "behave the same" on all types σ that might be used for α. This has far-reaching consequences, in particular for modularity and data abstraction. As we will see in the next lecture, if a client to a library that hides an implementation type is *parametric* in this type, then the library implementer or maintainer has the opportunity to replace the implementation with a different one without risk of breaking the client code.

The informal idea that a function behaves parametrically in a type variable α is surprisingly difficult to capture technically. Reynolds [Rey83] realized that it must be done *relationally*. For example, a function $f : \forall\alpha.\,\alpha \to \alpha$ is parametric if for any two types τ and σ, and any relation between values of type τ and σ, if we pass f related arguments it will return related results. This oversimplifies the situation somewhat, but it may provide the right intuition. What Reynolds showed is that in a polymorphic λ-calculus with products and Booleans, all expressions are parametric.

We begin by considering how to define different practically useful notions of equality since, ultimately, parametricity will allow us to prove program equalities.

2 Extensional Equality

When reasoning about equality so far, mostly as we were trying to assess if two functions witness a type isomorphism, we have been using a "simple" extensional equality. This requires knowledge about their types. We write $e \equiv e' : \tau$ when two closed expressions $e : \tau$ and $e' : \tau$ are *equal at type* τ. We use the intuition behind Kleene equality to say that either both diverge (have no value), or they both reduce to equal values.

(expressions) $e \equiv e' : \tau$ iff both e and e' diverge (that is, have no value), or $e \mapsto^* v$ and $e' \mapsto^* v'$ with $v = v' : \tau$.

To compare values for equality, the definition depends on the type. If it is a type at which values are *observable* we examine the structure of the value and compare the components at their appropriate types.

($\times$) $v = v' : \tau_1 \times \tau_2$ iff $v = \langle v_1, v_2 \rangle$ and $v' = \langle v_1', v_2' \rangle$ for some v_1, v_2, v_1', v_2' and $v_1 = v_1' : \tau_1$ and $v_2 = v_2' : \tau_2$.

(1) $v = v' : 1$ iff $v = \langle\,\rangle = v'$.

(+) $v = v' : \sum_{i \in I}(i : \tau_i)$ iff $v = j \cdot v_j$ and $v' = j \cdot v_j'$ for some j, v_j and v_j' with $v_j = v_j' : \tau_j$.

(ρ?) $v = v' : \rho\alpha.\,\tau$ iff $v = \mathsf{fold}\ v_1$ and $v' = \mathsf{fold}\ v_1'$ and $v_1 = v_1' : [\rho\alpha.\,\tau/\alpha]\tau$.

In all cases except for the last the type at which two values are compared becomes smaller, but the value becomes smaller in the last case which can sometimes help to still make sense of the definition. This is the case when the type $\rho\alpha.\,\tau$ is *purely positive*, which means it is entirely constructed from $\times$, 1, $+$, and ρ.

It becomes even more interesting when types are *not observable*, like functions or lazy pairs. In those cases we apply extensionality. For example, we say two functions are equal if they return equal results when applied to the same value.

($\to$) $v = v' : \tau_1 \to \tau_2$ iff for all $v_1 : \tau_1$ we have $v\,v_1 \equiv v'\,v_1 : \tau_2$.

Two short notes: (1) because we are in a call-by-value language, we quantify here over *values* v_1, and (2) we do not match against the shape of v and v' (which we could, by the canonical forms theorem) but instead probe its behavior via the elimination rule, comparing the resulting expressions. Similarly, two lazy pairs are equal if their projections are equal.

(&) $v = v' : \tau_1 \mathbin{\&} \tau_2$ iff $\mathsf{fst}\ v \equiv \mathsf{fst}\ v'$ and $\mathsf{snd}\ v \equiv \mathsf{snd}\ v'$.

A difficulty arises with polymorphic types. Consider

(∀?) $v = v' : \forall\alpha.\,\tau$ iff for all closed types σ we have $v\,[\sigma] = v'\,[\sigma] : [\sigma/\alpha]\tau$.

The problem here is that among the possible types σ we find $\forall\alpha.\,\tau$ itself, which means that the definition would be circular. This is similar to the issue with recursive types, but there we had a way out by considering purely positive types only. For polymorphism, that does not help. However, we could *stratify* the language of types, as is done in languages such as ML or Haskell. For example, in ML the type α with so-called *prefix polymorphism* the type variable α can be instantiated only with quantifier-free types. In the next section we see another more general way out which also allows us to capture the notion of parametricity.

We can also observe that the clauses in the definition of equality except the (discarded) one for universal quantification depend only on components of the given type. So we can easily consider subsets or future extensions of the language without changing the structure of the definition. We would like to preserve this positive property while repairing the issue with polymorphic types.

3 Logical Equality

The notion of extensional equality (and the underlying Kleene equality) are almost sufficient, but it is insufficient when we come to *parametricity*. The problem is that we want to compare expressions not at the same, but at related types. This means, for example, that in comparing e and e' and type $\forall\alpha.\,\tau$ we cannot apply e and e' to the exact *same* type σ. Instead, we must apply them to *related* types. This in turn means that the two expressions we are comparing may not have the same type but *related* types. The notion of equality we derive from this is called *logical equality* because it is based on *logical relations* [Sta85], one of the many connections between logic and computation. We write

$$e \approx e' \in [\![\tau]\!]$$

if the expressions e and e' stand in the relation designated by τ. This is a slight abuse of notation because, as we will see, τ can be more than just a type. Also, we no longer require that e and e' should have type τ. For the reason explained above, they may not have the same type. Furthermore, they may not even be well-typed anymore which allows a richer set of applications for logical equality. We also have a second relation, designated by $[\tau]$ that applies only to values. We write $v \sim v' \in [\tau]$ if the values v and v' are related by $[\tau]$. We define

(expressions) $e \approx e' \in [\![\tau]\!]$ iff $e \mapsto^* v$ and $e' \mapsto^* v'$ and $v \sim v' \in [\tau]$.

We assume here, to keep the development simple, that all expressions terminate. The clauses for the positive types remain essentially the same as for extensional equality, where we restrict recursive types to be purely positive.

($\times$) $v = v' \in [\tau_1 \times \tau_2]$ iff $v = \langle v_1, v_2 \rangle$ and $v' = \langle v_1', v_2' \rangle$ for some v_1, v_2, v_1', v_2' and $v_1 = v_1' \in [\tau_1]$ and $v_2 = v_2' \in [\tau_2]$.

(1) $v = v' \in [1]$ iff $v = \langle\,\rangle = v'$.

(+) $v = v' \in [\sum_{i \in I}(i : \tau_i)]$ iff $v = j \cdot v_j$ and $v' = j \cdot v_j'$ for some j, v_j and v_j' with $v_j = v_j' \in [\tau_j]$.

(ρ^+) $v = v' \in [\rho\alpha^+ .\, \tau^+]$ iff $v = \mathsf{fold}\ v_1$ and $v' = \mathsf{fold}\ v_1'$ and $v_1 = v_1' \in [[\rho\alpha^+ .\, \tau^+/\alpha^+]\tau^+]$.

To be explicit, we define the purely positive types as

$$\tau^+ ::= \tau_1^+ \times \tau_2^+ \mid 1 \mid \sum_{i \in I}(i : \tau_i^+) \mid \rho\alpha^+ .\, \tau^+ \mid \alpha^+$$

Even though the type becomes larger in the last clause, the definition is not circular because the values we are comparing get smaller.

Even the case for lazy pairs mirror what we had before.

(&) $v \sim v' \in [\tau_1 \,\&\, \tau_2]$ iff $\mathsf{fst}\ v \approx \mathsf{fst}\ v' \in [\![\tau_1]\!]$ and $\mathsf{snd}\ v \approx \mathsf{snd}\ v' \in [\![\tau_2]\!]$

The definition becomes different when we come to universal quantification, where we need to be careful to (a) avoid circularity in the definition, and (b) capture the idea behind parametricity. We write $R : \sigma \leftrightarrow \sigma'$ for a relation between values $v : \sigma$ and $v' : \sigma'$, and $v\ R\ v'$ if R relates v and v'. In some situation when we would like to reason about parametricity using logical relations, we may need to put some conditions on R, but here we think of it as an arbitrary relation on values. We then define

($\forall$) $v \sim v' \in [\forall\alpha.\, \tau]$ iff for all closed types σ and σ' and relations $R : \sigma \leftrightarrow \sigma'$ we have $v[\sigma] \approx v'[\sigma'] \in [\![[R/\alpha]\tau]\!]$

(R) $v \sim v' \in [R]$ iff $v\ R\ v'$.

The second clause here is a new base case in the definition of $[\tau]$, in addition to the type 1. It is needed because we substitute an arbitrary relation R for the type variable α in the clause for universal quantification. So when we encounter R we just use it to compare v and v'.

For functions, we apply them to *related* arguments and check that their results are again *related*.

($\rightarrow$) $v \sim v' \in [\tau_1 \rightarrow \tau_2]$ iff for all $v_1 \sim v_1' \in [\tau_1]$ we have $v\, v_1 \approx v'\, v_1' \in [\![\tau_2]\!]$

We have taken a big conceptual step, because what we write as type τ actually now contains relations instead of type variables, as well as ordinary type constructors.

The quantification structure should make it clear that logical equality in general is difficult to establish. It requires a lot: for two arbitrary types and an arbitrary relation between values, we have to establish properties of e and e'. It is an instructive exercise to check that

$$\Lambda\alpha.\, \lambda x.\, x \sim \Lambda\alpha.\, \lambda x.\, x \in [\forall\alpha.\, \alpha \rightarrow \alpha]$$

To check: $\Lambda\alpha.\, \lambda x.\, x \sim \Lambda\alpha.\, \lambda x.\, x \in [\forall\alpha.\, \alpha \rightarrow \alpha]$
This holds if $\lambda x.\, x \approx \lambda x.\, x \in [\![R \rightarrow R]\!]$ for arbitrary σ, σ' and $R : \sigma \leftrightarrow \sigma'$
This holds if $\lambda x.\, x \sim \lambda x.\, x \in [R \rightarrow R]$
This holds if $(\lambda x.\, x)\, v_1 \approx (\lambda x.\, x)\, v_1' \in [\![R]\!]$ for arbitrary $v_1 \sim v_1' \in [R]$
This holds if $v_1 \sim v_1' \in [R]$, which is true by assumption

There is nothing wrong with this proof, but let's turn this reasoning around and present it in the "forward" direction, just to see it in a different form.

Let σ, σ', $R : \sigma \leftrightarrow \sigma'$ be arbitrary	Assumption
$v_1\ R\ v_1'$ for some arbitrary v_1 and v_1'	Assumption
$v_1 \sim v_1' \in [R]$	By defn. of $\sim$ at $[R]$
$(\lambda x.\, x)\, v_1 \approx (\lambda x.\, x)\, v_1' \in [\![R]\!]$	By defn. of $\approx$ at $[\![R]\!]$
$\lambda x.\, x \sim \lambda x.\, x \in [R \rightarrow R]$	By defn. of $\sim$ at $[R \rightarrow R]$
	since v_1 and v_1' were arbitrary
$\lambda x.\, x \approx \lambda x.\, x \in [\![R \rightarrow R]\!]$	By defn. of $\approx$ at $[\![R \rightarrow R]\!]$
$\Lambda\alpha.\, \lambda x.\, x \sim \Lambda\alpha.\, \lambda x.\, x$	By defn. of $\sim$ at $[\forall\alpha.\, \alpha \rightarrow \alpha]$
	since R was arbitrary

Conversely, we can imagine that *knowing* that two expressions are parametrically equal is very powerful, because we can instantiate this with arbitrary types σ and σ' and relations between them. The *parametricity theorem* now states that all well-typed expressions are related to themselves.

Theorem 1 (Parametricity [Rey83]) *If* $\cdot\,;\cdot \vdash e : \tau$ *then* $e \approx e : \tau$

We will not go into the proof of this theorem, but just explore its consequences.

4 Some Useful Properties

In a couple of places we will use the following properties, which follow directly from small-step determinacy and the definition of $[\![\tau]\!]$.

(Closure under Expansion) If $e \approx e' \in [\![\tau]\!]$ and $e_0 \mapsto^* e$ and $e'_0 \mapsto^* e'$ then $e_0 \approx e'_0 \in [\![\tau]\!]$.

(Closure under Reduction) If $e \approx e' \in [\![\tau]\!]$ and $e \mapsto^* e_0$ and $e' \mapsto^* e'_0$ then $e_0 \approx e'_0 \in [\![\tau]\!]$.

Also, the call-by-value strategy entails the following properties for reasoning about logical equality.

(Closure under Application) If $e_1 \approx e'_1 \in [\![\tau_2 \to \tau_1]\!]$ and $e_2 \approx e'_2 \in [\![\tau_2]\!]$ then $e_1\, e_2 \approx e'_1\, e'_2 \in [\![\tau_2]\!]$.

(Closure under Type Application) If $e \approx e' \in [\![\forall\alpha.\,\tau]\!]$ and $R : \sigma \leftrightarrow \sigma'$ then $e[\sigma] \approx e'[\sigma'] \in [\![[R/\alpha]\tau]\!]$.

5 Exploiting Parametricity

Parametricity allows us to deduce information about functions knowing only their (polymorphic) types. For example, with only terminating functions, the type

$$f : \forall\alpha.\,\alpha \to \alpha$$

for a value f implies that f is (logically) equivalent to the identity function

$$f \sim \Lambda\alpha.\,\lambda x.\,x \in [\forall\alpha.\,\alpha \to \alpha]$$

Let's prove this. Unfortunately, the first few steps are the "difficult" direction of the parametricity.

By definition, this means to show that

For every pair of types σ *and* σ' *and relation* $R : \sigma \leftrightarrow \sigma'$*, we have* $f\,[\sigma] \approx (\Lambda\alpha.\,\lambda x.\,x)\,[\sigma'] \in [\![R \to R]\!]$

Now fix arbitrary σ, σ' and R. By definition of logical equivalence at $R \to R$, this holds iff

For all $v_0 \sim v_0' \in [R]$ *we have* $f\,[\sigma]\,v_0 \approx (\lambda\alpha.\,\lambda x.\,x)\,[\sigma']\,v_0' \in [\![R]\!]$

By definition of logical equality at R, this is equivalent to showing that

$v_0 \; R \; v_0'$ *implies* $f[\sigma]\,v_0 \mapsto^* w$, $(\Lambda\alpha.\,\lambda x.\,x)[\sigma']\,v_0' \mapsto^* w'$ *and* $w \; R \; w'$.

By the rules of evaluation, this is the case if and only if

$f[\sigma]\,v_0 \mapsto^* w_0$ and $w_0 \; R \; v_0'$, assuming $v_0 \; R \; v_0'$

So our proof would be complete if we could show that $f[\sigma']\,v_0 \mapsto^* v_0$. To prove this, we exploit the parametricity of f (by the parametricity theorem), using the a well-chosen relation S.

$f \sim f \in [\forall\alpha.\,\alpha \to \alpha]$ *by parametricity.*

Now define a new relation $S : \sigma \leftrightarrow \sigma$ such that $v_0 \; S \; v_0$ for the specific v_0 from the first half of the argument. Then

$f[\sigma] \approx f[\sigma] \in [\![S \to S]\!]$ *by definition of* $\sim$ *at polymorphic type.*

Applying the definition of logical equality at function type and the assumption that $v_0 \; S \; v_0$ we conclude

$f[\sigma]\,v_0 \approx f[\sigma]\,v_0 \in [\![S]\!]$

which is the same as saying

$f[\sigma]\,v_0 \mapsto^* w_0$ *and* $w_0 \; S \; w_0$

By definition, S only relates v_0 to itself, so $w_0 = v_0$.

$w_0 = v_0$ *and therefore* $f[\sigma]\,v_0 \mapsto^* v_0$

This completes the proof.

Similar proofs show, for example, that $f : \forall\alpha.\,\alpha \to \alpha \to \alpha$ must be equal to the first or second projection function. It is instructive to reason through the details of such arguments, but we move on to a different style of example.

6 Theorems for Free!

A slightly different style of application of parametricity is laid out in Philip Wadler's *Theorems for Free!* [Wad89]. Let's see what we can derive from

$$f : \forall\alpha.\, \alpha \to \alpha$$

for a value f. First, parametricity tells us

$$f \sim f \in [\forall\alpha.\, \alpha \to \alpha]$$

This time, we pick types τ and τ' and a relation R which is in fact a function $R : \tau \to \tau'$. Evaluation of R has the effect of closing the corresponding relation under Kleene equality. Then

$$f[\tau] \approx f[\tau'] \in [\![R \to R]\!]$$

Now, for arbitrary values $v : \tau$ and $v' : \tau'$, $v\ R\ v'$ actually means $R\, v \mapsto^* v'$. Using the definition of $\sim$ at function type we get

$$f[\tau]\, v \approx f[\tau']\, (R\, v) \in [\![R]\!]$$

but this in turn means

$$R\,(f[\tau]\, v) \mapsto^* w \quad \text{and} \quad f[\tau']\,(R\, v) \mapsto^* w \quad \text{for some value } w$$

Wadler summarizes this by stating that for any function $R : \tau \to \tau'$,

$$R \circ f[\tau] = f[\tau'] \circ R$$

that is, f commutes with any function R. If τ is non-empty and we have $v_0 : \tau$ and choose $\tau' = \tau$ and $R = \lambda x.\, v_0$ we obtain

$$\begin{array}{lcl} R\,(f[\tau]\, v_0) & \mapsto^* & v_0 \\ f[\tau]\,(R\, v_0) & \mapsto^* & f[\tau]\, v_0 \end{array}$$

so we find $f[\tau]\, v_0 \mapsto^* v_0$ which, since v_0 was arbitrary, is another way of saying that f behaves like the identity function.

Exercises

Exercise 1 Prove that $\forall\alpha.\, \alpha \to \alpha \cong 1$. You may use the results of Sections 3 and Section 5.

Exercise 2 Prove, using parametricity, that if we have $f : \forall\alpha.\, \alpha \to \alpha \to \alpha$ for a value f then either $f \sim \Lambda\alpha.\, \lambda x.\, \lambda y.\, x \in [\forall\alpha.\, \alpha \to \alpha \to \alpha]$ or $f \sim \Lambda\alpha.\, \lambda x.\, \lambda y.\, y \in [\forall\alpha.\, \alpha \to \alpha \to \alpha]$.

Exercise 3 Prove, using parametricity, that there cannot be a closed value $f : \forall\alpha.\, \alpha$.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

[Sta85] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[Wad89] Philip Wadler. Theorem for free! In J. Stoy, editor, *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359, London, UK, September 1989. ACM.