Lecture Notes on Parametricity

15-814: Types and Programming Languages
Frank Pfenning

Lecture 12
October 11, 2018

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 a future 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 the 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 is 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 Kleene Equality

The most elementary nontrivial notion of equality just requires that expressions are equal if they evaluate to the same value. We write $e \simeq e'$ (e is *Kleene-equal* to e') if either $e \mapsto^* v$ and $e' \mapsto^* v$ for some value v, or e and e' both diverge.

For the remainder of this lecture we assume that all expressions terminate, that is, evaluate to a value. This means we cannot permit arbitrary recursive types (due to the shallow embedding of the untyped λ-calculus) or arbitrary recursive expressions. We will not be precise about possible syntactic restrictions or extensions in the study of parametricity, but you may consult the given sources for details.

How far does Kleene equality go? For Booleans, for example, it works very well because $e \simeq e' : \mathit{bool}$ is quite sensible: two Boolean expressions are equal if they both evaluate to *true* or they both evaluate to *false*. Similarly, $e \simeq e' : \mathit{nat}$ is the appropriate notion of equality: two expressions of type *nat* are equal if they evaluate to the same natural number.

We can construct bigger types for which Kleene equality still has the right meaning. For example, expressions of type $\mathit{bool} \otimes \mathit{nat}$ should be equal if they evaluate to the same value, which will be in fact a pair of two values whose equality we already understand.

The following so-called *purely positive types* all have *fully observable values*, so Kleene equality equates *exactly* those expressions we would like to be equal.

$$\text{Purely positive types} \quad \tau^+ \quad ::= \quad \tau_1^+ \otimes \tau_2^+ \mid 1 \mid \tau_1^+ + \tau_2^+ \mid 0 \mid \rho\alpha^+.\,\tau^+ \mid \alpha^+$$

With *negative types*, namely $\tau \to \sigma$ or $\tau \mathbin{\&} \sigma$ this is no longer the case. The problem is that we assumed we cannot directly observe the body of a function (which is an arbitrary expression). So, even though intuitively the function on Booleans that doubly-negates its argument and the identity function should be equal. We write $\cong$ for this stronger notion of equality.

$$\lambda x.\, \mathit{not}\,(\mathit{not}\,x) \cong \lambda x.\, x : \mathit{bool} \to \mathit{bool}$$

Another way to express this situation is that we would like to consider functions *extensionally*, via their input/output relationship, but not their definition. There are other aspects of these two functions that are not equal. For example, the identity function has many other types, while the double-negation does not. The identity function is likely to be more efficient. And the former may lose some points in a homework assignment on functional

programming because it is less elegant than the latter. Similarly, a function performing bubble sort is extensionally equivalent to one performing quicksort, while otherwise they have many different characteristics.

We ignore intensional aspects of functions in our extensional notions of equality in this lecture. Keeping this in mind, a reasonable approach would be to define

($\to$) $e \cong e' : \tau_1 \to \tau_2$ iff for all $v_1 : \tau_1$ we have $e\, v_1 \cong e'\, v_1 : \tau_2$

(&) $e \cong e' : \tau_1 \,\&\, \tau_2$ iff $e \cdot l \cong e' \cdot l : \tau_1$ and $e \cdot r \cong e' \cdot r : \tau_2$

With this definition we can now easily prove that the two Boolean functions above are extensionally equal. The key is to distinguish the cases of $v_1 = \mathit{true}$ and $v_1 = \mathit{false}$ for $v_1 : \mathit{bool}$, which follows from the canonical form theorem.

3 Logical Equality

The notions of Kleene equality and extensional equality are almost sufficient, but when we come to parametricity the extensional equality as sketched so does not function correctly any more. 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 $\tau_1 \to \tau_2$ we cannot apply e and e' to the exact *same* value. Instead, we must apply it to *related* values. The second clause for lazy pairs can remain the same. We write $e \sim e' : \tau$ for this refined notion. It is called *logical equality* because it is based on *logical relations*, one of the many connections between logic and computation.

($\to$) $e \sim e' : \tau_1 \to \tau_2$ iff for all $v_1 \sim v_1' : \tau_1$ we have $e\, v_1 \sim e'\, v_1' : \tau_2$

(&) $e \sim e' : \tau_1 \,\&\, \tau_2$ iff $e \cdot l \sim e' \cdot l : \tau_1$ and $e \cdot r \sim e' \cdot r : \tau_2$

We can also fill in the definitions for positive type constructors. Because their values are directly observable, we just inspect their form and compare the component values.

($+$) $e \sim e' : \tau_1 + \tau_2$ iff either $e \mapsto^* l \cdot v_1$, $e \mapsto^* l \cdot v_1'$ and $v_1 \sim v_1' : \tau_2$ or $e \mapsto^* r \cdot v_2$, $e \mapsto^* r \cdot v_2'$ and $v_2 \sim v_2' : \tau_2$.

(0) $e \sim e' : 0$ never.

($\otimes$) $e \sim e' : \tau_1 \otimes \tau_2$ iff $e \mapsto^* \langle v_1, v_2 \rangle$ and $e' \mapsto^* \langle v_1', v_2' \rangle$ and $v_1 \sim v_1' : \tau_1$ and $v_2 \sim v_2' : \tau_2$

(1) $e \sim e' : 1$ iff $e \mapsto^* \langle\,\rangle$ and $e' \mapsto^* \langle\,\rangle$.

A key aspect of this notion of equality is that it is defined by induction over the structure of the type, which can easily be seen by examining the definitions. We always reduce the question of equality at a type to its components (assuming there are any). This is also the reason why recursive types are excluded, even though large classes of recursive types (in particular, *inductive* and *coinductive* types) can be included systematically.

The question for this lecture is how to extend it to include parametric polymorphism. The straightforward approach

$$e \sim e' : \forall\alpha.\,\tau \text{ iff for all closed } \sigma,\ e \sim e' : [\sigma/\alpha]\tau \text{ ?}$$

fails because the type $[\sigma/\alpha]\tau$ may contain $\forall\alpha.\,\tau$. Moreover, parametric functions are supposed to map related values at related types to related results, and this definition does not express this. Instead, we write $R : \sigma \leftrightarrow \sigma'$ for a relation between expressions $e : \sigma$ and $e' : \sigma'$, and $e\ R\ e'$ if R relates e and e'. Furthermore we require R to be *admissible*, which means it is closed under Kleene equality.[1] That is, if $f \simeq e$, $e \sim e'$, and $e' \simeq f'$ then also $f \sim f'$. Now we define

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

(*R*) $e \sim e' : R$ with $e : \tau$, $e' : \tau'$ and $R : \tau \leftrightarrow \tau'$ iff $e\ R\ e'$.

This is a big conceptual step, because what we write as type τ actually now contains admissible relations instead of type variables, as well as ordinary types constructors. Because Kleene equality itself is admissible (it's trivially closed under Kleene equality) we can instantiate α with Kleene equality on the same type σ. A base case of the inductive definitions is then ordinary Kleene equality.

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 admissible relation, we have to establish properties of e and e'. It is an instructive exercise to check that

$$\lambda x.\,x \sim \lambda x.\,x : \forall\alpha.\,\alpha \to \alpha$$

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.

[1] Other admissibility conditions are possible, depending on the application.

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

What we suggested you tediously prove by hand above is an immediate consequence of this theorem.

4 Exploiting Parametricity

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

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

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

$$f \sim \lambda x.\, x : \forall\alpha.\, \alpha \to \alpha$$

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* τ *and* τ' *and admissible relation* $R : \tau \leftrightarrow \tau'$,
> *we have* $f \sim \lambda x.\, x : R \to R$

Now fix arbitrary τ, τ' and R. Next, we use the definition of logical equivalence at type $\tau \to \tau'$ to see that this is equivalent to

> *For every pair of values* $v_0 \sim v_0' : R$ *we have* $f\, v_0 \sim (\lambda x.\, x)\, v_0' : R$

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

> $v_0\ R\ v_0'$ *implies* $f\, v_0\ R\ (\lambda x.\, x)\, v_0'$

Since R is closed under Kleene equality this is the case if and only if

> $f\, v_0\ R\ v_0'$ assuming $v_0\ R\ v_0'$

This is true if $f\, v_0 \mapsto^* v_0$ since R is closed under Kleene equality.

So our theorem is complete if we can show that $f\, v_0 \mapsto^* v_0$. To prove this, we use the parametricity theorem with a well-chose relation. We start with

> $f \sim f : \forall\alpha.\, \alpha \to \alpha$ *by parametricity.*

Now define the new relation $S : \tau \leftrightarrow \tau$ such that $v_0\ S\ v_0$ for the specific v_0 from the first half of the argument and close it under Kleene equality. Then

$f \sim f : S \to S$ *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

$$f\, v_0 \sim f\, v_0 : S$$

which is the same as saying

$$f\, v_0\ S\ f\, v_0$$

By definition, S only relates expressions that are Kleene-equal to v_0, so

$$f\, v_0 \mapsto^* v_0$$

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.

5 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

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

First, parametricity tells us

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

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

$$f \sim f : R \to R$$

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

$$f\, x \sim f\, (R\, x) : R$$

but this in turn means

$$R\, (f\, x) \simeq f\, (R\, x)$$

This means, for any function $R : \tau \to \tau'$,

$$R \circ f \simeq f \circ R$$

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

$$\begin{array}{rcl} R\,(f\,v_0) & \simeq & v_0 \\ f\,(R\,v_0) & \simeq & f\,v_0 \end{array}$$

so we find $f\,v_0 \simeq v_0$ which, since v_0 was arbitrary, is another way of saying that f is equivalent to the identity function.

For more interesting examples, we extend the notion of logical equivalence to lists. Since lists are inductively defined, we can call upon a general theory to handle them, but since we haven't discussed this theory we give the specific definition.

(τ list) $e \sim e' : \tau$ **list** iff $e \mapsto^* [v_1, \ldots, v_n]$, $e' \mapsto^* [v'_1, \ldots, v'_n]$ and $v_i \sim v'_i : \tau$ for all $1 \leq i \leq n$.

The example(s) are easier to understand if we isolate the special case R **list** for an admissible relation $R : \tau \to \tau'$ which is actually a function. In this case we obtain

$e \sim e' : R$ **list** for an admissible $R : \tau \to \tau'$ iff $(\mathit{map}\ R)\,e \simeq e'$.

Here, $\mathit{map} : (\tau \to \tau') \to (\tau\ \mathbf{list} \to \tau'\ \mathbf{list})$ is the usual mapping function with

$$(\mathit{map}\ R)\,[v_1, \ldots, v_n] \mapsto^* [R\,v_1, \ldots, R\,v_n]$$

Returning to examples, what can the type tell us about a function

$$f : \forall \alpha.\, \alpha\ \mathbf{list} \to \alpha\ \mathbf{list}?$$

If the function is parametric, it should not be able to examine the list elements, or create new ones. However, it should be able to drop elements, duplicate elements, or rearrange them. We will try to capture this equationally, just following our nose in using parametricity to see what we end up at.

We start with

$f \sim f : \forall \alpha.\, \alpha\ \mathbf{list} \to \alpha\ \mathbf{list}$ *by parametricity.*

Now let $R : \tau \to \tau'$ be an admissible relation that's actually a function. Then

$f \sim f : R\ \mathbf{list} \to R\ \mathbf{list}$ *by definition of* $\sim$.

Using the definition of $\sim$ on function types, we obtain

For any $l : \tau\ \mathbf{list}$ *and* $l' : \tau\ \mathbf{list}$ *with* $l\ (R\ \mathbf{list})\ l'$ *we have* $f\ l\ (R\ \mathbf{list})\ f\ l'$

By the remark on the interpretation of R **list** when R is a function, this becomes

If $(map\ R)\ l \simeq l'$ *then* $(map\ R)\ (f\ l) \simeq f\ l'$

or, equivalently,

$(map\ R)\ (f\ l) \simeq f\ ((map\ R)\ l)$.

In short, f commutes with *map* R. This means we can either map R over the list and then apply f to the result, or we can apply f first and then map R over the result. This implies that f could not, say, make up a new element v_0 not in l. Such an element would occur in the list returned by the right-hand side, but would occur as $R\,v_0$ on the left-hand side. So if we have a type with more than one element we can choose R so that $R\,v_0 \neq v_0$ (like a constant function) and the two sides would be different, contradicting the equality we derived.

We can use this equation of improve efficiency of code. For example, if we know that f might reduce the number of elements in the list (for example, skipping every other element), then mapping R over the list after the elements have been eliminated is more efficient than the other way around. Conversely, if f may duplicate some elements then it would be more efficient to map R over the list first and then apply f. The equality we derived from parametricity allows this kind of optimization.

We have, however, to be careful when nonterminating functions may be involved. For example, if R diverges on an element v_0 then the two sides may not be equal. For example, f might drop v_0 from the list l so the right-hand side would diverge while the left-hand side would have a value.

Here are two other similar results provided by Wadler [Wad89].

$$f : \forall\alpha.\ (\alpha\ \mathbf{list})\ \mathbf{list} \to \alpha\ \mathbf{list}$$
$$(map\ R)\ (f\ l) \simeq f\ ((map\ (map\ R))\ l)$$

$$f : \forall\alpha.\ (\alpha \to bool) \to \alpha\ \mathbf{list} \to \alpha\ \mathbf{list}$$
$$(map\ R)\ (f\ (\lambda x.\ p\ (R\,x))\ l) \simeq f\ p\ ((map\ R)\ l)$$

These theorems do not quite come "for free", but they are fairly straightforward consequences of parametricity, keeping in mind the requirement of termination.

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.

[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.