Lecture Notes on Parametricity

15-814: Types and Programming Languages
Jan Hoffmann

Lecture 16
Tuesday, October 29, 2024

1 Introduction

Recall our previous discussion of parametric polymorphism in System F (also called the Polymorphic λ-Calculus). Parametric polymorphism is the idea that a function of type $\forall(t.t \to \tau)$ will "behave the same" on all types σ that might be used for t. 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 t is surprisingly difficult to capture technically. Reynolds [Rey83] realized that it must be done *relationally*. For example, a function $g : \forall(t.t \to t)$ is parametric if for any two types τ and σ, and any relation between values of type τ and σ, if we pass g related arguments it will return related results.

Let's illustrate this idea with another (unknown) function

$$\cdot \vdash f : \forall(t.t \to t \to t)$$

Assume f is *parametric* in its type argument. We have

$$\begin{array}{lcl} f[\texttt{bool}] & : & \texttt{bool} \to \texttt{bool} \to \texttt{bool} \\ f[\texttt{nat}] & : & \texttt{nat} \to \texttt{nat} \to \texttt{nat} \end{array}$$

Now consider a relation $R : \texttt{bool} \leftrightarrow \texttt{nat}$ between Booleans and natural numbers such that *false* $R\ \overline{0}$ and *true* $R\ \overline{n}$ for $n > 0$. If

$$f[\texttt{bool}](\mathit{false})(\mathit{true}) \mapsto^* \mathit{false}$$

then it must also be the case that, for example,

$$f[\texttt{nat}](\overline{0})(\overline{17}) \mapsto^* \overline{0}$$

On the other hand, from the indicated behavior and relation we cannot immediately make a statement about

$$f[\texttt{nat}](\overline{42})(\overline{0})$$

But we can pick a different relation! Let *false* $S\ \overline{42}$ and *true* $S\ \overline{0}$ (and no other values are related). From the relation S and parametricity we conclude

$$f[\texttt{nat}](\overline{42})(\overline{0}) \mapsto^* \overline{42}$$

We can see that parametricity is quite powerful, since we can tell a lot about the behavior of f without knowing its definition. A detail that we avoided discussing is that f is in fact a higher-order function and, say, $f[\texttt{nat}]$ maps natural numbers to functions of type $\texttt{nat} \to \texttt{nat}$. So what does parameterized actually state for such a function? As we will see, it states that for any relation $R : \texttt{bool} \leftrightarrow \texttt{nat}$, if we apply $f[\texttt{bool}]$ to and $f[\texttt{nat}]$ two related arguments then we get two functions that map R-related arguments to R-related results. This is why we needed to include $(\mathit{true}, \overline{0})$ in the relations R and S in the example.

What Reynolds showed is that in a polymorphic λ-calculus with products and Booleans, all expressions are parametric in this sense.

Parametricity, arises from the definition of equality for expressions. So we start by exploring equality for System T and then consider an extension of System T with type abstraction and application.

2 Equality for System T

Recall the types of System T.

$$\begin{array}{lcll} \mathsf{Typ} & \tau & ::= & \texttt{nat} \\ & & & \tau_1 \to \tau_2 \end{array}$$

We will not refer to the definition of the expressions in this lecture but repeat the definition for completeness.

			Abstract	Concrete
Exp	e	$::=$	x	x
			$\mathtt{z}$	$\mathtt{z}$
			$\mathtt{s}(\mathrm{e})$	$\mathtt{s}(\mathrm{e})$
			$\mathtt{rec}\{e_0; x.y.e_1\}(e)$	$\mathtt{rec}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\, \mathtt{with}\, y \hookrightarrow e_1\}$
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\, (x : \tau)\, e$
			$\mathtt{ap}(e_1; e_2)$	$e_1(e_2)$

How can we define equality for the expressions of System T? We will only consider programs, that is, closed well-typed expressions, in this lecture. However, the concepts generalize to well-typed open expressions and actually have to be generalized to prove the theorems that we will mention. The generalizations rely on concepts such as *closing substitutions* that you have worked with in the homework assignments.

Observational Equality The most basic definition of equality is called *observational equality* or *contextual equality*. It states programs e and e' are equal if they can be interchanged in any program e_0 without changing the result of the evaluation of e_0. This is idea is reflected in the following more formal definition.

Definition 1 (Observational Equality) *Two expression e and e' are observationally equal, written $e \cong e'$, if $e : \tau$, $e' : \tau$, and for all expressions e_c such $x : \tau \vdash e_c : \mathtt{nat}$ we have*

$$[e/x]e_c \mapsto^* \overline{n} \quad \textit{iff} \quad [e'/x]e_c \mapsto^* \overline{n}$$

Note that observational equality is symmetric, reflexive, and transitive. The generalization to open expression is intuitive but requires a bit of work and we are not going to discuss it. You can find it in Chapter 47.1 of PFPL.

Observational equality is not specific to System T but can be defined for basically all programming languages in a similar way. It is agnostic to the language features, which can include non-termination and other side effects. The reason we are only considering contexts e_c with result type `nat` is that it allows us to reduce the notion of equality of programs, which is complex, to equality of natural numbers, which is well understood and can be inductively defined.

Logical Equality (Call-By-Value) Observational equality is both generic and intuitive. It also has mathematical properties that make it a canonical notion of equality (see PFPL). However, observational equality has the disadvantage that it does not directly lead to a strategy for proving that two programs e and e' are equal. We would have to consider all possible contexts e_c and compare the programs $[e/x]e_c$ and $['e/x]e_c$. Instead, it would be desirable to have an inductive definition of equality that we can apply to obtain a derivation for the equality of two expressions. It turns out that it is difficult (and probably impossible) to define such a notion of equality using inference rules.

However, we can define a *logical equality* by induction on the type of the expressions we compare. This approach is similar to the one we took to show the canonicity theorem for System T. Logical equality is indeed also considered a logical relation. However, there are some differences to the *candidate* relation that we defined for canonicity. One difference is that we work in a call-by-value setting in this lecture. As a result, we define two separate relations for values and expressions.

$e \approx e' : \tau$	programs e and e' of type τ are logically equal
$v \sim v' : \tau$	closed values v and v' of type τ are logically equal

For values, we define $v \sim v' : \tau$ by induction on τ. Natural numbers are logically equal if they are equal according to the standard equality for natural numbers. Functions are logically equal if they map logically equal expressions to logically equal results. Expressions are logically equal if they evaluate to logically equal values.

Natural numbers: $v \sim v' : \mathtt{nat}$ if $v = v'$.

Functions: $v \sim v' : \tau_1 \to \tau_2$ if $v_1 \sim v_1' : \tau_1$ implies $v(v_1) \approx v'(v_1') : \tau_2$ for all values $v_1 : \tau_1$ and $v_1' : \tau_1$.

Expressions: $e \approx e' : \tau$ if $e \mapsto^* v$, $e' \mapsto^* v'$, and $v \sim v' : \tau$.

We can show that logical equality coincides with observational equality. This is the fundamental theorem of our logical relation, which means that we defined it so that this theorem holds. The proof is not trivial and requires a generalization of the definitions to open expressions. It can be found in Chapter 46 (Equality of System T) of PFPL.

Theorem 1 (Fundamental Theorem) *Let $e : \tau$ and $e' : \tau$. Then $e \cong e'$ if and only if $e \approx e' : \tau$.*

Definition of Logical Relations During the lecture, the question arose what exactly a logical relation is. The short answer is that it is not entirely clear. An attempt at a definition is that it is a relation over expressions that is defined inductively on the type structure. However, as we will see in the next section, not even this general definition is entirely accurate.

It is probably best to think about logical relations as relations on expressions that are defined to have some fundamental property or for which we can prove a fundamental theorem. Logical relations are often defined on the type structure but sometimes we need to extend the definition beyond types (as in the next section).

In this course, you have so far seen two logical relations. On Homework 3, we defined a unary logical relation that related a single expression to a realizer. We then showed that every expressions has a realizer, which was the fundamental theorem. This was the key to our goal of proving canonicity or termination of programs of type `nat`. Here, we defined a binary logical relation that relates two expressions. We defined this relation so that it coincides with observational equality. Our goal was to give an inductive definition of equality so that we can use induction to show that two expressions are equal.

3 Equality and Polymorphism: Parametricity

We now consider System F+, which is a combination of System T and System F. We extend System T as follows.

$$\begin{array}{llllllll}
\mathsf{Typ} & \tau & ::= & \texttt{nat} & \qquad \mathsf{Exp} & e & ::= & \ldots \\
 & & & \tau_1 \to \tau_2 & & & & \Lambda(t)\, e \\
 & & & t & & & & e\,[\tau] \\
 & & & \forall(t.\tau) & & & &
\end{array}$$

Observational Equality We work with System F+ instead of System F because the base type `nat` enables us to use the exact same definition for *observational equality* as for System T. In System F, we could attempt to use the Church encoding for natural numbers as the base type. However, since numbers are polymorphic functions in this encoding, we would not be able to reduce equality to a well understood notion.

Logical Equality To define logical equality for System F+, let us attempt to extend the definition of logical equality for System T. To this end, we have to define $v \sim v' : \forall(t.\tau)$. Inspired by the definition for functions, we try to following.

Attempt: $v \sim v' : \forall(t.\tau)$ if for all closed types σ $v[\sigma] \approx v[\sigma'] : [\sigma/t]\tau$.

However, because of the impredicativity of polymorphic types, this leads to a cyclic definition. The problem is that we can substitute "large" types σ for t. As a concrete example, consider $v \sim v' : \forall(t.t)$. Then

$$v \sim v' : \forall(t.t) \quad \text{if} \quad \ldots v \sim v' : [\forall(t.t)/t]t$$

And since $[\forall(t.t)/t]t = \forall(t.t)$, this is cyclic definition.

Recall from our discussion of parametricity that a function f is parametric if it preserves relations between different types. Our solution for avoiding a cyclic definition is to use such relations in the definition of logical equality. For closed type τ and τ', we write

$$R : \tau \leftrightarrow \tau' \quad \text{if} \quad R \subseteq \{(v, v') \mid v : \tau, v \ \mathsf{val}, v' : \tau', v' \ \mathsf{val}\}$$

Now, we define equality with respect to a relation $R : \tau \leftrightarrow \tau'$ as follows.

Relations: $v \sim v' \in [R]$ if $v \, R \, v'$

We lift this definition to closed types that can contain relations.

$$\begin{array}{lcl} \mathsf{Typ} \quad \tau & ::= & \texttt{nat} \\ & & \tau_1 \to \tau_2 \\ & & t \\ & & \forall(t.\tau) \\ & & R \end{array}$$

The cases for natural numbers, functions, and expressions are similar to the same cases for System T.

Natural numbers: $v \sim v' \in [\texttt{nat}]$ if $v = v'$.

Functions: $v \sim v' \in [\tau_1 \to \tau_2]$ if $v_1 \sim v_1' \in [\tau_1]$ implies $v(v_1) \approx v'(v_1') \in [\tau_2]$ for all values $v_1 : \tau_1$ and $v_1' : \tau_1$.

Expressions: $e \approx e' \in [\tau]$ if $e \mapsto^* v$, $e' \mapsto^* v'$, and $v \sim v' \in [\tau]$.

For type abstraction, the issue with impredicativity is solved by only substituting relations (a base case) for type variables. (It follows from the definition of $R : \sigma \leftrightarrow \sigma'$ that σ and σ' are closed types.)

Type abstractions: $v \sim v' \in [\forall(t.\tau)]$ if for all relations $R : \sigma \leftrightarrow \sigma'$, we have $v[\sigma] \approx v'[\sigma'] \in [[R/t]\tau]$.

Parametricity We can now formulate the parametricity theorem.

Theorem 2 (Parametricity) *If $e : \tau$ then $e \approx e \in [\tau]$.*

For the proof, we have to generalize the theorem to open expressions. The proof then proceeds by induction on the judgment $\Gamma \vdash e : \tau$.

One application of the parametricity theorem is the justification of the definition of logical equality for System F+.

Theorem 3 (Fundamental Theorem) *Let $e : \tau$ and $e' : \tau$. Then $e \cong e'$ if and only if $e \approx e' : \tau$.*

The proof can be found in Chapter 48 of PFPL. It requires a generalization of equality to open expressions.

References

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