Lecture Notes on Theorems for Free

15-814: Types and Programming Languages
Frank Pfenning

Lecture 15
Tue Oct 28, 2025

1 Introduction

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. We looked at the unary case in the last lecture, which was useful for termination, and is similarly useful to prove type soundness. However, it does not let us compare two functions to establish that they have observably equivalent behavior. Reynolds [1983] 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. As an example, let's consider some (unknown) function

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

and assume it *parametric* in its type argument. We have

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

Even though we started with the same function f, at this point the two expression have *different type*, yet we would like to express that they nevertheless behave the same.

Now consider a relation R such that $\mathit{false}\ R\ \overline{0}$ and $\mathit{true}\ R\ \overline{n}$ for $n > 0$. If

$$f\,[\mathit{bool}]\ \mathit{false}\ \mathit{true} \mapsto^* \mathit{false}$$

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

$$f\,[\mathit{nat}]\ \overline{0}\ \overline{17} \mapsto^* \overline{0}$$

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

$$f\,[\mathit{nat}]\ \overline{42}\ \overline{0}$$

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

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

We can see that parametricity is quite powerful, since we can tell a lot about the behavior of f without knowing its definition

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

We begin by recalling extensional equality and then a new form of equality based on the idea of parametricity called *logical equality*.

2 Extensional Equality Revisited

In Lecture 7 we defined an *extensional equality* between expressions for pairs, units, sums, and functions. The purpose was to reason about type isomorphisms. We repeat the definition here, and discuss one additional case. First, expressions are simply evaluated to values that are then compared with a more specialized relation.

Expressions: $e \approx e' : \tau$ iff $e \mapsto^* v$, $e' \mapsto^* v'$ with v, v' values, and $v \sim v' : \tau$ or both e and e' diverge.

For positive types (eager pairs, sums, unit) we compare the structure of the values (which are observable), while for negative types (functions, lazy pairs) we apply the destructor and then compare the results.

Functions: $v \sim v' : \tau_1 \to \tau_2$ iff for all $v_1 : \tau_1$ we have $v\ v_1 \approx v'\ v_1 : \tau_2$.

Pairs: $v \sim v' : \tau_1 \times \tau_2$ iff $v = \langle v_1, v_2 \rangle$, $v' = \langle v_1', v_2' \rangle$ and $v_1 \sim v_1' : \tau_1$ and $v_2 \sim v_2' : \tau_2$.

Units: $v \sim v' : 1$ iff $v = \langle\,\rangle$ and $v' = \langle\,\rangle$ (which is always the case, by the canonical forms theorem).

Sums: $v \sim v' : \sum_{i \in I}(i : \tau_i)$ iff $v = k \cdot v_k$ and $v' = k \cdot v_k'$ and $v_k \sim v_k' : \tau_k$ for some $k \in I$.

Lazy Pair: $v \sim v' : \tau_1 \,\&\, \tau_2$ iff $v.\pi_1 \approx v'.\pi_1 : \tau_1$ and $v.\pi_2 \approx v'.\pi_2 : \tau_2$

One question is how to extend this to polymorphic and recursive types. For example, we might want to say

Universal Quantification(?): $v \sim v' : \forall\alpha.\,\tau$ iff for all closed σ we have $v\,[\sigma] \approx v'[\sigma] : [\sigma/\alpha]\tau$.

However, this is problematic since the type on the right-hand side of this definition, $[\sigma/\alpha]\tau$ could be much bigger than the original type $\forall\alpha.\,\tau$. In fact, it could contain the original type. So it seems possible be trapped into a chain of reasoning such as

$$v \sim v' : \forall\alpha.\,\tau \to \tau \quad \text{iff} \quad \ldots \quad \text{iff} \quad v \sim v' : \forall\alpha.\,\tau \to \tau$$

so the equality may somehow not be well-defined. This is a consequence of the *impredicative* nature of $\forall\alpha.\,\tau$ because the universal quantifier includes the new type we are trying to define. Many logicians and type theorists reject such definitions as circular. Fortunately, this problem is by now familiar, as is the solution. This solution also solves the second problem, namely that we are locked in to comparing only expressions of the same type.

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 *same* type σ. In fact, in this binary setting types denote *binary relations beween values*, and, by extension, expressions. So a type variable α should stand for an arbitrary *relation* between values! The notion of equality we derive from this is called *logical equality* because it is based on *logical relations* [Statman, 1985], 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 $v \sim v' \in [\tau]$ for some values v and v'.

The clauses for the observable types remain essentially the same as for extensional equality, but omitting recursive types for now.

Pairs: $v \sim v' \in [\tau_1 \times \tau_2]$ 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 $v_1 \sim v_1' \in [\tau_1]$ and $v_2 \sim v_2' \in [\tau_2]$.

Unit: $v \sim v' \in [1]$ iff $v = \langle\,\rangle = v'$.

Sums: $v \sim v' \in [\sum_{i\in I}(i : \tau_i)]$ iff $v = k \cdot v_k$ and $v' \sim k \cdot v_k'$ for some k, v_k and v_k' with $v_k \sim v_k' \in [\tau_k]$.

The case for lazy pairs mirrors what we had before, using the destructors.

Lazy Pairs: $v \sim v' \in [\tau_1 \,\&\, \tau_2]$ iff $v.\pi_1 \approx v'.\pi_2 \in [\![\tau_1]\!]$ and $v.\pi_2 \approx v'.\pi_2 \in [\![\tau_2]\!]$

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. In some situations 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

Universal Quantification: $v \sim v' \in [\forall\alpha.\,\tau]$ iff for all relations R between closed values we have $v[R] \approx v'[R] \in [\![[R/\alpha]\tau]\!]$

(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'.

As in the last lecture, if we want to avoid mixing syntax and semantics in this manner, we can parameterize the relation by a substitution of relations for its type variables.

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. An important observation about this definition is that the type *does* get smaller because the clause for the relation R is a base case. So the issue with impredicativity with the attempt to define a simple extensional equality vanishes!

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

Functions: $v \sim v' \in [\tau_1 \to \tau_2]$ iff for all $v_1 \sim v_1' \in [\tau_1]$ we have $v\, v_1 \approx v'\, v_1' \in [\![\tau_2]\!]$

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

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

To check: $\Lambda\alpha.\, \lambda x.\, x \sim \Lambda\alpha.\, \lambda x.\, x \in [\forall\alpha.\, \alpha \to \alpha]$
This holds if $(\Lambda\alpha.\, \lambda x.\, x)\, [R] \approx (\Lambda\alpha.\, \lambda x.\, x)\, [R] \in [\![R \to R]\!]$ for arbitrary R
This holds if $\lambda x.\, x \sim \lambda x.\, x \in [R \to R]$
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 R be an arbitrary relation between closed values	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 $[\![R]\!]$
$\lambda x.\, x \sim \lambda x.\, x \in [R \to R]$	By defn. of $\sim$ at $[R \to R]$
	since v_1 and v_1' were arbitrary
$(\Lambda\alpha.\, \lambda x.\, x)\, [R] \approx (\Lambda\alpha.\, \lambda x.\, x)\, [R] \in [\![R \to R]\!]$	By defn. of $\approx$ at $[\![R \to R]\!]$
$\Lambda\alpha.\, \lambda x.\, x \sim \Lambda\alpha.\, \lambda x.\, x$	By defn. of $\sim$ at $[\forall\alpha.\, \alpha \to \alpha]$
	since σ, σ', and R were 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* also called *fundamental theorem of logical relations* now states that all well-typed expressions are related to themselves. This property holds in a language without general recursive types and general fixed point expressions.

Theorem 1 (Parametricity [Reynolds, 1983]) *If* $\cdot \vdash e : \tau$ *then* $e \approx e \in [\![\tau]\!]$

We will not go into the proof of this theorem, but most of the ideas are familiar from the last two lectures. We generalize over contexts and relations between them and the proceed by rule induction of the given typing derivation. Besides the original paper, there are a number of proofs in the literature including in the textbook [Harper, 2016, Chapter 48] in a language and formalization that's quite similar to ours. We do not go into detail under which conditions it might be restored in the presence of recursive types and fixed point expressions (see, for example, Ahmed [2006]).

4 Some Useful Properties

In a couple of places we may use the following properties, which follow directly from small-step determinism (sequentiality) 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 $e_1\, e_2 \approx e_1'\, e_2' \in [\![\tau_1]\!]$.

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

5 Exploiting Parametricity

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

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

implies that f behaves like the identity function! We express this first by proving

$$f\ [\sigma_0]\ v_0 \mapsto^* v_0 \quad \text{for all types } \sigma_0 \text{ and values } v_0 : \sigma_0$$

Later, we prove this property in a second form, namely that f is logically equivalent to the polymorphic identity.

For simplicity, assume f is a value. By the parametricity theorem, we have

$$f \approx f \in [\![\forall\alpha.\, \alpha \to \alpha]\!]$$

By definition of $[\![-]\!]$ and the fact that f is a value, we obtain

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

By definition of $[\forall\alpha.-]$, this entails that

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

for any relation R on closed values. In view of the property we want to show, we pick R_0 such that $v_0\ R_0\ v_0$ for any value $v_0 : \sigma_0$. That is, R_0 relates only v_0 to itself and not any other values. This mean we have

$$f\ [R_0] \approx f\ [R_0] \in [\![R_0 \to R_0]\!]$$

Next, by definition of $[\![-]\!]$ we find $f\ [R_0] \mapsto^* f_0$ for some value f_0 and

$$f_0 \sim f_0 \in [R_0 \to R_0]$$

By definition of $[\_ \to \_]$ this means that for any value v such that $v \sim v \in [R_0]$ we have $f_0\, v \approx f_0\, v \in [R_0]$. We pick $v = v_0$ because $v_0\ R_0\ v_0$ and consequently also

$$v_0 \sim v_0 \in [R_0]$$

Therefore we conclude

$$f_0\, v_0 \approx f_0\, v_0 \in [\![R_0]\!]$$

Again, by definition of $[\![-]\!]$ we know that $f_0\, v_0 \mapsto^* w$ for some w with

$$w \sim w \in [R_0]$$

which in turn is the case if and only if

$$w\ R_0\ w$$

by the definition of $[R_0]$. But R_0 was chosen so it relates only v_0 to v_0, so we conclude that

$$w = v_0$$

Unwinding the chain of evaluations under our call-by-value strategy we find

$$f\ [\sigma_0]\ v_0 \mapsto^* f_0\ v_0 \mapsto^* w = v_0$$

and our theorem is proved.

Out next goal is to show that any value $f : \forall\alpha.\, \alpha \to \alpha$ 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 R we have $f\ [R] \approx (\Lambda\alpha.\, \lambda x.\, x)\ [R] \in [\![R \to R]\!]$*

Because $(\Lambda\alpha.\, \lambda x.\, x)\ [R] \mapsto \lambda x.\, x$, our desired conclusion holds if $f\ [R] \mapsto^* f_0$ for some value f_0 and

$$f_0 \sim \lambda x.\, x \in [R \to R]$$

Applying the definition of $[\_ \to \_]$, this is true if

> *For all $v \sim v' \in [R]$ we have $f_0\, v \approx (\lambda x.\, x)\, v' \in [\![R]\!]$*

So assume $v \sim v' \in [R]$. It remains to show that

> *$f_\sigma\, v \mapsto^* w$ for some w with $w \sim v' \in [R]$.*

By the previous argument (starting from the parametricity of f) we know that $f_0\, v \mapsto^* v$, so determinism gives us $w = v$. Then $w\ R\ v'$ follows from $v\ R\ v'$ and $w = v$.

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. We turn our attention to a whole class of such consequences of parametricity called "*theorems for free*" [Wadler, 1989].

6 Theorems for Free!

A slightly different style of application of parametricity is laid out in Wadler's [1989] *Theorems for Free!*. The idea is that the for the arbitrary relation we actually pick a *well-typed function*!

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* $R : \tau \to \tau'$. Then

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

which means that $f[R] \mapsto^* f_0$

$$f_0 \sim f_0 \in [R \to R]$$

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

$$f_0\,v \approx f_0\,(R\,v) \in [\![R]\!]$$

but this in turn means

$$R\,(f_0\,v) \mapsto^* w \quad \text{and} \quad f_0\,(R\,v) \mapsto^* w \quad \text{for some value } w$$

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. Here, he uses τ and τ' instead of R since we assume R is well-typed, relating values of type τ to values of type τ'. If τ is non-empty and we have $v_0 : \tau$ and choose $\tau' = \tau$ and $R = \lambda x.\, v_0$ we obtain

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

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. We have now reached this conclusion a few different ways.

7 Parametricity on Lists

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. Here, we think of lists defined with

$$\mathsf{list}\ \tau = \mu\alpha.\, (\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \alpha)$$

even though type constructors like list haven't been formally introduced into our language. Then we use a shorthand notation for lists, that is, elaborate the left-hand side into the right-hand side:

$$\langle\!\langle e_1, \ldots, e_n \rangle\!\rangle \triangleq \mathsf{fold}\ \mathbf{cons} \cdot \langle e_1, \ldots \mathsf{fold}\ \mathbf{cons} \cdot \langle e_n, \mathsf{fold}\ \mathbf{nil} \cdot \langle\,\rangle \rangle \rangle$$

We then extend the notion of logical equalities to values of list type *inductively* over the structure of the list, which reduces the type of the relation because each element has a smaller type.

$$v \sim v' \in [\mathsf{list}\ \tau] \textit{ iff } v = [v_1, \ldots, v_n],\ v' = [v'_1, \ldots, v'_n] \textit{ and } v_i \sim v'_i \in [\tau] \textit{ for all } 1 \le i \le n.$$

Then we have, for example, a polymorphic *map* function:

$$\begin{array}{l} \mathit{map} : \forall\alpha.\, \forall\beta.\, (\alpha \to \beta) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \beta \\ \mathit{map} = \Lambda\alpha.\, \Lambda\beta.\, \mathsf{fix}\ m.\, \lambda f.\, \lambda l. \\ \qquad \mathsf{case}\ (\mathsf{unfold}\ l)\ \ (\ \mathbf{nil} \cdot \langle\,\rangle \Rightarrow \mathsf{fold}\,\mathbf{nil} \cdot \langle\,\rangle \\ \qquad\qquad\qquad\qquad \mid \mathbf{cons} \cdot \langle x, l' \rangle \Rightarrow \mathsf{fold}\,\mathbf{cons} \cdot \langle f\ x, m\ f\ l' \rangle\ ) \end{array}$$

The map function then satisfies (for $f : \tau \to \tau'$):

$$\mathit{map}\ [\tau]\ [\tau']\ f\ [v_1, \ldots, v_n] = [f\ v_1, \ldots, f\ v_n]$$

where equality here is Kleene equality (both sides reduce to the same value). The example(s) are easier to understand if we isolate the special case list R for a relation $R : \tau \to \tau'$ which is actually a function. In this case we obtain

$$v \sim v' \in [\mathsf{list}\ R] \text{ for an } R : \tau \to \tau' \text{ iff } (\mathit{map}\ [\tau]\ [\tau']\ R)\ v = v'.$$

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

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

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

$$f \sim f \in [\forall\alpha.\, \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha] \textit{ by parametricity.}$$

Now let $R : \tau \to \tau'$ be a function. Then $f[R] \mapsto^* f_0$ and

$$f_0 \sim f_0 \in [\mathsf{list}\ R \to \mathsf{list}\ R] \textit{ by definition of } \approx.$$

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

$$\textit{For any values } l : \mathsf{list}\ \tau \textit{ and } l' : \mathsf{list}\ \tau' \textit{ with } l\ (R\ \mathsf{list})\ l' \textit{ we have } f_0\ l\ (R\ \mathsf{list})\ f_0\ l'$$

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

$$\textit{If } (\mathit{map}\ [\tau]\ [\tau']\ R)\ l = l' \textit{ then } (\mathit{map}\ [\tau]\ [\tau']\ R)\ (f_0\ l) = f_0\ l'$$

or, replacing the relation R by its argument and result type as appropriate:

$$(\mathit{map}\ [\tau]\ [\tau']\ R)\ (f\ [\tau]\ l) = f\ [\tau']\ ((\mathit{map}\ [\tau]\ [\tau']\ R)\ l).$$

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 [1989].

$$f : \forall\alpha.\,(\alpha\ \mathsf{list})\ \mathsf{list} \to \alpha\ \mathsf{list}$$
$$(\mathit{map}\ [\tau]\,[\tau']\,R)\,(f\,[\tau]\,l) = f\ [\tau']\,((\mathit{map}\ [\mathsf{list}\,\tau]\,[\mathsf{list}\,\tau']\,(\mathit{map}\ [\tau]\,[\tau']\,R))\ l)$$

$$f : \forall\alpha.\,(\alpha \to \mathit{bool}) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$$
$$(\mathit{map}\ [\tau]\,[\tau']\,R)\ (f\,[\tau]\,(\lambda x.\,p\,(R\,x))\,l) = f\,[\tau']\,p\,((\mathit{map}\,[\tau]\,[\tau']\,R)\,l)$$

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

8 Inductive Types

The theorems in the preceding section treat lists as a new primitive type. On the other hand, we can define every instance list τ directly as a recursive type, so it is worth considering if we can find a more general characterization of a class of recursive types to which parametricity applies. These are *inductive types* whose values are defined purely inductively. We call these here *fully observable types*. Logical equality then does not have to reference computation at all and write τ^+. It is not a coincidence that we previously introduced this class as the types whose values are directly *observable*. To be explicit, we define the purely positive types as

$$\text{Purely positive types}\quad \tau^+ \quad ::= \quad \tau_1^+ \times \tau_2^+ \mid 1 \mid \textstyle\sum_{i\in I}(i : \tau_i^+) \mid \mu\alpha^+.\,\tau^+ \mid \alpha^+$$

Then we can define:

Recursion: $v \sim v' \in [\mu\alpha^+.\,\tau^+]$ iff $v = \mathsf{fold}\ v_1$ and $v' = \mathsf{fold}\ v_1'$ and $v_1 \sim v_1' \in [[\mu\alpha^+.\,\tau^+/\alpha^+]\tau^+]$.

Even though the type becomes larger in the last clause, the definition is not circular because the values we are comparing get smaller. In fact, we can prove that $v \sim v' \in [\tau^+]$ iff $v = v'$. So the clauses for positive types are mostly useful if negative types are embedded in them.

This characterization means that we can obtain "theorems for free" for free for functions operating over (polymorphic) purely positive types such as lists, trees, and similar structures.

Exercises

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

Exercise 2 Prove, using parametricity, that there cannot be a closed value $f : \forall\alpha.\,\alpha$.

Exercise 3 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 4 Prove, using parametricity, that $\forall\alpha.\, \alpha \to \alpha \to \alpha \cong 2$.

References

Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In P. Sestoft, editor, *15th European Symposium on Programming (ESOP 2006)*, pages 69–83, Vienna, Austria, March 2006. Springer LNCS 3924.

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

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

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

Philip Wadler. Theorems 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.