Theorems for free!

Philip Wadler
University of Glasgow*

Abstract

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

1 Introduction

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

The purpose of this paper is to explain the trick. But first, let's look at an example.

Say that r is a function of type

$$r : \forall X.\ X^* \to X^*.$$

Here X is a type variable, and X^* is the type "list of X". From this, as we shall see, it is possible to conclude that r satisfies the following theorem: for all types A and A' and every total function $a : A \to A'$ we have

$$a^* \circ r_A = r_{A'} \circ a^*.$$

Here $\circ$ is function composition, and $a^* : A^* \to A'^*$ is the function "map a" that applies a elementwise to a list of A yielding a list of A', and $r_A : A^* \to A^*$ is the instance of r at type A.

The intuitive explanation of this result is that r must work on lists of X for *any* type X. Since r is provided with no operations on values of type X, all it can do is rearrange such lists, independent of the values contained in them. Thus applying a to each element of a list and then rearranging yields the same result as rearranging and then applying a to each element.

For instance, r may be the function $reverse : \forall X.\ X^* \to X^*$ that reverses a list, and a may be the function $code : Char \to Int$ that converts a character to its ASCII code. Then we have

$$\begin{array}{rl} & code^*\ (reverse_{Char}\ [\text{'a'}, \text{'b'}, \text{'c'}]) \\ = & [99, 98, 97] \\ = & reverse_{Int}\ (code^*\ [\text{'a'}, \text{'b'}, \text{'c'}]) \end{array}$$

which satisfies the theorem. Or r may be the function $tail : \forall X.\ X^* \to X^*$ that returns all but the first element of a list, and a may be the function $inc : Int \to Int$ that adds one to an integer. Then we have

$$\begin{array}{rl} & inc^*\ (tail_{Int}\ [1,2,3]) \\ = & [3,4] \\ = & tail_{Int}\ (inc^*\ [1,2,3]) \end{array}$$

which also satisfies the theorem.

On the other hand, say r is the function $odds : Int^* \to Int^*$ that removes all odd elements from a list of integers, and say a is inc as before. Now we have

$$\begin{array}{rl} & inc^*\ (odds_{Int}\ [1,2,3]) \\ = & [2,4] \\ \neq & [4] \\ = & odds_{Int}(inc^*\ [1,2,3]) \end{array}$$

and the theorem is *not* satisfied. But this is not a counterexample, because $odds$ has the wrong type: it is too specific, $Int^* \to Int^*$ rather than $\forall X.\ X^* \to X^*$.

This theorem about functions of type $\forall X.\ X^* \to X^*$ is pleasant but not earth-shaking. What is more exciting is that a similar theorem can be derived for *every* type.

The result that allows theorems to be derived from types will be referred to as the *parametricity* result, because it depends in an essential way on parametric polymorphism (types of the form $\forall X.\ T$). Parametricity is just a reformulation of Reynolds' abstraction theorem: terms evaluated in related environments yield related values [Rey83]. The key idea is that types may be read

*Author's address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland. Electronic mail: wadler@cs.glasgow.ac.uk.

as relations. This result will be explained in Section 2 and stated more formally in Section 6.

Some further applications of parametricity are shown in Figure 1, which shows several types and the corresponding theorems. Each name was chosen, of course, to suggest a particular function of the named type, but the associated theorems hold for *any* function that has the same type (so long as it can be defined as a term in the pure polymorphic lambda calculus). For example, the theorem given for *head* also holds for *last*, and the theorem given for *sort* also holds for *nub* (see Section 3).

The theorems are expressed using operations on functions that correspond to operations on types. Corresponding to the list type A^* is the map operation a^* that takes the function $a : A \rightarrow A'$ into the function $a^* : A^* \rightarrow A'^*$. Similarly, corresponding to the product type $A \times B$ is the operation $a \times b$ that takes the functions $a : A \rightarrow A'$ and $b : B \rightarrow B'$ into the function $a \times b : A \times B \rightarrow A' \times B'$; it is defined by $(a \times b)\ (x, y) = (a\ x, b\ y)$. As we shall see, it will be necessary to generalise to the case where a, b, a^*, and $a \times b$ are relations.

How useful are the theorems so generated? Only time and experience will tell, but some initial results are encouraging:

- In general, the laws derived from types are of a form useful for algebraic manipulation. For example, many of the laws in Figure 1 allow one to "push map through a function".
- Three years ago I co-authored a paper on the derivation of an algorithm for compiling pattern-matching in functional languages [BW86]. The derivation used nine general theorems about higher-order functions such as map and sort. Of the nine theorems, five follow immediately from the types.
- Sheeran has developed a formal approach to the design of VLSI circuits that makes heavy use of mathematical laws. She has found that many of the laws she needs can be generated from types using the methods described here, and has already written a paper describing how to do so [She89].

Not surprisingly, using a more specific type system allows even more theorems to be derived from the type of a function; this has already been explored to a certain extent by Sheeran [She89]. So there is reason to believe that further research will further extend the applicability of this method.

Many functional languages, including Standard ML [Mil84,Mil87], Miranda[1] [Tur85], and Haskell [HW88], are based on the Hindley/Milner type system [Hin69,Mil78,DM82]. This system is popular because types need not be given explicitly; instead, the principal (most general) type of a function can be inferred from its definition. However, for the purposes of this paper it is more convenient to use the Girard/Reynolds type system [Gir72,Gir86,Rey74,Rey83] (also known as the polymorphic lambda calculus, the second order lambda calculus, and System F). In the Girard/Reynolds system it is necessary to give the types of bound variables explicitly. Further, if a function has a polymorphic type then type applications must be explicitly indicated. This is done via subscripting; for example, the instance of the function $r : \forall X.\ X^* \rightarrow X^*$ at the type A is written $r_A : A^* \rightarrow A^*$.

Every program in the Hindley/Milner system can automatically be translated into one in the Girard/Reynolds system. All that is required is a straightforward modification of the type inference algorithm to decorate programs with the appropriate type information. On the other hand, the inverse translation is not always possible, because the Girard/Reynolds system is more powerful than Hindley/Milner.

Both the Hindley/Milner and the Girard/Reynolds system satisfy the *strong normalisation* property: every term has a normal form, and every reduction sequence leads to this normal form. As a corollary, it follows that the fixpoint operator,

$$fix : \forall X.\ (X \rightarrow X) \rightarrow X$$

cannot be defined as a term in these systems. For many purposes, we can get along fine without the fixpoint operator, because many useful functions (including all those shown in Figure 1) may be defined in the Girard/Reynolds system without its use. Indeed, every recursive function that can be proved total in second-order Peano arithmetic can be written as a term in the Girard/Reynolds calculus [FLO83,Gir72,GLT89]. This includes, for instance, Ackerman's function (see [Rey85]), but it excludes interpreters for most languages (including the Girard/Reynolds calculus itself).

If the power of unbounded recursion is truly required, then *fix* can be added as a primitive. However, adding fixpoints weakens the power of the parametricity theorem. In particular, if fixpoints are allowed then the theorems in Figure 1 hold in general only when the functions a and b are strict (that is, when $a\ \bot = \bot$ and $b\ \bot = \bot$)[2]. For this reason, the bulk of this paper assumes that fixpoints are not provided; but the necessary adjustment to allow fixpoints is described in Section 7.

[1]Miranda is a trademark of Research Software Limited.

[2]This is similar to the restriction to strict coercion functions in [BCGS89], and is adopted for a similar reason.

Assume $a : A \rightarrow A'$ and $b : B \rightarrow B'$.

$$head : \forall X.\ X^* \rightarrow X$$
$$a \circ head_A = head_{A'} \circ a^*$$

$$tail : \forall X.\ X^* \rightarrow X^*$$
$$a^* \circ tail_A = tail_{A'} \circ a^*$$

$$(+\!\!+) : \forall X.\ X^* \rightarrow X^* \rightarrow X^*$$
$$a^*\ (xs +\!\!+_A ys) = (a^*\ xs) +\!\!+_{A'} (a^*\ ys)$$

$$concat : \forall X.\ X^{**} \rightarrow X^*$$
$$a^* \circ concat_A = concat_{A'} \circ a^{**}$$

$$fst : \forall X.\ \forall Y.\ X \times Y \rightarrow X$$
$$a \circ fst_{AB} = fst_{A'B'} \circ (a \times b)$$

$$snd : \forall X.\ \forall Y.\ X \times Y \rightarrow Y$$
$$b \circ snd_{AB} = snd_{A'B'} \circ (a \times b)$$

$$zip : \forall X.\ \forall Y.\ (X^* \times Y^*) \rightarrow (X \times Y)^*$$
$$(a \times b)^* \circ zip_{AB} = zip_{A'B'} \circ (a^* \times b^*)$$

$$filter : \forall X.\ (X \rightarrow Bool) \rightarrow X^* \rightarrow X^*$$
$$a^* \circ filter_A\ (p' \circ a) = filter_{A'}\ p' \circ a^*$$

$$sort : \forall X.\ (X \rightarrow X \rightarrow Bool) \rightarrow X^* \rightarrow X^*$$
$$\text{if for all } x, y \in A,\ \ (x < y) = (a\ x <' a\ y) \text{ then}$$
$$a^* \circ sort_A\ (<) = sort_{A'}\ (<') \circ a^*$$

$$fold : \forall X.\ \forall Y.\ (X \rightarrow Y \rightarrow Y) \rightarrow Y \rightarrow X^* \rightarrow Y$$
$$\text{if for all } x \in A, y \in B,\ \ b\ (x \oplus y) = (a\ x) \otimes (b\ y) \text{ and } b\ u = u' \text{ then}$$
$$b \circ fold_{AB}\ (\oplus)\ u = fold_{A'B'}\ (\otimes)\ u' \circ a^*$$

$$I : \forall X.\ X \rightarrow X$$
$$a \circ I_A = I_{A'} \circ a$$

$$K : \forall X.\ \forall Y.\ X \rightarrow Y \rightarrow X$$
$$a\ (K_{AB}\ x\ y) = K_{A'B'}\ (a\ x)\ (b\ y)$$

Figure 1: Examples of theorems from types

The fundamental idea of parametricity is not new. A restricted version of it appears in Reynolds' original paper on the polymorphic lambda calculus [Rey74], where it is called the representation theorem, and a version similar to that used here appears in [Rey83], where it is called the abstraction theorem. Other versions include the logical relations of Mitchell and Meyer [MM85,Mit86]; and the dinatural transformations of Bainbridge, Freyd, Girard, Scedrov, and Scott [BFSS87,FGSS88], from whom I have taken the name "parametricity".

So far as I am aware, all uses of parametricity to date have been "general": they say something about possible implementations of the polymorphic lambda calculus (e.g. that the implementation is correct independent of the representation used) or about its models (e.g. that models should only be allowed that satisfy parametricity). The main contribution of this paper is to suggest that parametricity also has "specific" applications: it says interesting things about particular functions with particular types[3].

A second contribution of this paper is to present an updated proof of the abstraction theorem. The proof given here is based on that in [Rey83]. Unfortunately, that proof is expressed in terms of a "naive" set-theoretic model of the polymorphic lambda calculus; Reynolds later proved that such models do not exist [Rey84]. Fortunately, the proof adapts easily to the *frame models* of Bruce, Meyer, and Mitchell [BM84,MM85], and that is the approach taken in this paper. (For other models of the polymorphic lambda calculus, see [BTC88,Mes89,Pit87].)

The characterisation of parametricity given in this paper can be formulated more concisely in terms of category theory, where it can be re-expressed in terms of lax natural transformations. This will be the subject of a further paper.

The remainder of this paper is organised as follows. Sections 2 and 3 present the main new results: Section 2 presents the parametricity theorem, and Section 3 gives further applications. Sections 4–6 fill in the formalities: Section 4 describes the syntax of the polymorphic lambda calculus, Section 5 shows how its syntax can be given using frame models, and Section 6 gives the full statement of the parametricity theorem. Section 7 shows how the parametricity theorem should be adjusted to account for languages that use the fixpoint operator.

Acknowledgements. I am grateful to Harold Simmons for helping to formulate and prove the result about *map* in Section 3.5, and to Samson Abramsky, Val Breazu-Tannen, Peter Freyd, John Hughes, John Launchbury, John Reynolds, Andre Scedrov, and Mary Sheeran for their comments on this work.

[3]Since this paper was written, I have learned that Peter deBruin has recently discovered similar applications [deB89], and that John Reynolds already knew of the application in Section 3.8.

2 Parametricity explained

The key to extracting theorems from types is to read types as relations. This section outlines the essential ideas, using a naive model of the polymorphic lambda calculus: types are sets, functions are set-theoretic functions, etc. The approach follows that in [Rey83].

Cognoscenti will recognise a small problem here—there are no naive set-theoretic models of polymorphic lambda calculus! (See [Rey84].) That's ok; the essential ideas adopt easily to frame models [BM84,MM85]. This section sticks to the simple but naive view; the i's will be dotted and the t's crossed in Sections 4–6, which explain the same notions in the context of frame models.

The usual way to read a type is as a set. The type *Bool* corresponds to the set of booleans, and the type *Int* corresponds to the set of integers. If A and B are types, then the type $A \times B$ corresponds to a set of pairs drawn from A and B (the cartesian product), the type A^* corresponds to the set of lists with elements in A, and the type $A \rightarrow B$ corresponds to a set of functions from A to B. Further, if X is a type variable and $A(X)$ is a type depending on X, then the type $\forall X.\ A(X)$ corresponds to a set of functions that take a set B and return an element in $A(B)$.

An alternative is to read a type as a relation. If A and A' are sets, we write $\mathcal{A} : A \Leftrightarrow A'$ to indicate that a is a relation between A and A', that is, that $\mathcal{A} \subseteq A \times A'$. If $x \in A$ and $x' \in A'$, we write $(x, x') \in \mathcal{A}$ to indicate that x and x' are related by $\mathcal{A}$. A special case of a relation is the identity relation $I_A : A \Leftrightarrow A$, defined by $I_A = \{(x,x) \mid x \in A\}$. In other words, if $x, x' \in A$, then $(x, x') \in I_A$ iff $x = x'$. More generally, any function $a : A \rightarrow A'$ may also be read as a relation $\{(x, a\ x) \mid x \in A\}$. In other words, if $x \in A$ and $x' \in A'$, then $(x, x') \in a$ iff $a\ x = x'$.

To read types as relations, we give a relational equivalent for constant types and for each of the type constructors $A \times B$, A^*, $A \rightarrow B$, and $\forall X.\ A(X)$. Constant types, such as *Bool* and *Int*, may simply be read as identity relations, $I_{Bool} : Bool \Leftrightarrow Bool$ and $I_{Int} : Int \Leftrightarrow Int$.

For any relations $\mathcal{A} : A \Leftrightarrow A'$ and $\mathcal{B} : B \Leftrightarrow B'$, the relation $\mathcal{A} \times \mathcal{B} : (A \times B) \Leftrightarrow (A' \times B')$ is defined by

$$((x,y),(x',y')) \in \mathcal{A} \times \mathcal{B}$$
$$\text{iff}$$
$$(x,x') \in \mathcal{A} \text{ and } (y,y') \in \mathcal{B}.$$

That is, pairs are related if their corresponding components are related. In the special case where a and b are function, then $a \times b$ is the function defined by $(a \times b)\ (x, y) = (a\ x, b\ y)$.

For any relation $\mathcal{A} : A \Leftrightarrow A'$, the relation $\mathcal{A}^* : A^* \Leftrightarrow A'^*$ is defined by

$$([x_1, \ldots, x_n], [x'_1, \ldots, x'_n]) \in \mathcal{A}^* \\ \text{iff} \\ (x_1, x'_1) \in a \text{ and } \ldots \text{and } (x_n, x'_n) \in \mathcal{A}.$$

That is, lists are related if they have the same length and corresponding elements are related. In the special case where a is a function, a^* is the familiar "map" function defined by $a^*\ [x_1, \ldots, x_n] = [a\ x_1, \ldots, a\ x_n]$.

For any relations $\mathcal{A} : A \Leftrightarrow A'$ and $\mathcal{B} : B \Leftrightarrow B'$, the relation $\mathcal{A} \rightarrow \mathcal{B} : (A \rightarrow B) \Leftrightarrow (A' \rightarrow B')$ is defined by

$$(f, f') \in \mathcal{A} \rightarrow \mathcal{B} \\ \text{iff} \\ \text{for all } (x, x') \in \mathcal{A}, \quad (f\ x, f'\ x') \in \mathcal{B}.$$

That is, functions are related if they take related arguments into related results. In the special case where a and b are functions, the relation $a \rightarrow b$ will not necessarily be a function, but in this case $(f, f') \in a \rightarrow b$ is equivalent to $f' \circ a = b \circ f$.

Finally, we have to interpret $\forall$ as an operation on relations. Let $\mathcal{F}(\mathcal{X})$ be a relation depending on $\mathcal{X}$. Then $\mathcal{F}$ corresponds to a function from relations to relations, such that for every relation $\mathcal{A} : A \Leftrightarrow A'$ there is a corresponding relation $\mathcal{F}(\mathcal{A}) : F(A) \Leftrightarrow F'(A')$. Then the relation $\forall\mathcal{X}.\ \mathcal{F}(\mathcal{X}) : \forall X.\ F(X) \Leftrightarrow \forall X'.\ F'(X')$ is defined by

$$(g, g') \in \forall\mathcal{X}.\ \mathcal{F}(\mathcal{X}) \\ \text{iff} \\ \text{for all } \mathcal{A} : A \Leftrightarrow A', \quad (g_A, g'_{A'}) \in \mathcal{F}(\mathcal{A}).$$

That is, polymorphic functions are related if they take related types into related results. (Note the similarities in the definitions of $\mathcal{A} \rightarrow \mathcal{B}$ and $\forall\mathcal{X}.\ \mathcal{F}(\mathcal{X})$.)

Using the definitions above, any closed type T (one containing no free variables) can be read as a relation $\mathcal{T} : T \Leftrightarrow T$. The main result of this paper can now be described as follows:

> **Proposition.** *(Parametricity.) If t is a closed term of type T, then $(t, t) \in \mathcal{T}$, where $\mathcal{T}$ is the relation corresponding to the type T.*

A more formal statement of this result appears in Section 6, where it is extended to types and terms containing free variables.

3 Parametricity applied

This section first explains in detail how parametricity implies some of the theorems listed in the introduction and then presents some more general results.

3.1 Rearrangements

The result in the introduction is a simple consequence of parametricity. Let r be a closed term of type

$$r : \forall X.\ X^* \rightarrow X^*.$$

Parametricity ensures that

$$(r, r) \in \forall\mathcal{X}.\ \mathcal{X}^* \rightarrow \mathcal{X}^*.$$

By the definition of $\forall$ on relations, this is equivalent to

$$\text{for all } \mathcal{A} : A \Leftrightarrow A', \\ (r_A, r_{A'}) \in \mathcal{A}^* \rightarrow \mathcal{A}^*$$

By the definition of $\rightarrow$ on relations, this in turn is equivalent to

$$\text{for all } \mathcal{A} : A \Leftrightarrow A', \\ \text{for all } (xs, xs') \in \mathcal{A}^*, \\ (r_A\ xs, r_{A'}\ xs') \in \mathcal{A}^*$$

This can be further expanded in terms of the definition of $\mathcal{A}^*$. A more convenient version can be derived by specialising to the case where the relation $\mathcal{A}$ is a function $a : A \rightarrow A'$. The above then becomes

$$\text{for all } a : A \rightarrow A', \\ \text{for all } xs, \\ a^*\ xs = xs' \quad \text{implies} \quad a^*\ (r_A\ xs) = r_{A'}\ xs'$$

or, equivalently,

$$\text{for all } a : A \rightarrow A', \\ a^* \circ r_A = r'_A \circ a^*.$$

This is the version given in the introduction.

3.2 Fold

The function *fold* has the type

$$\mathit{fold} : \forall X.\ \forall Y.\ (X \rightarrow Y \rightarrow Y) \rightarrow Y \rightarrow X^* \rightarrow Y.$$

Parametricity implies that

$$(\mathit{fold}, \mathit{fold}) \in \forall\mathcal{X}.\ \forall\mathcal{Y}.\ (\mathcal{X} \rightarrow \mathcal{Y} \rightarrow \mathcal{Y}) \rightarrow \mathcal{Y} \rightarrow \mathcal{X}^* \rightarrow Y.$$

Let $a : A \rightarrow A'$ and $b : B \rightarrow B'$ be two functions. Applying the definition of $\forall$ on relations, twice, specialised to functions, gives

$$(\mathit{fold}_{AB}, \mathit{fold}_{A'B'}) \in (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow a^* \rightarrow b$$

Applying the definition of $\rightarrow$ on relations, twice, gives

$$\begin{array}{l} \text{for all } (\oplus, \oplus') \in (a \rightarrow b \rightarrow b), \\ \quad \text{for all } (u, u') \in b, \\ \quad\quad (\mathit{fold}_{AB}\ (\oplus)\ u, \mathit{fold}_{A'B'}\ (\oplus')\ u') \in a^* \rightarrow b. \end{array}$$

Here $(\oplus)$ is just the name of a function of two arguments; by the usual convention, $(\oplus)\ x\ y$ may be written in the infix form $x \oplus y$. Further expansion shows that the condition $(\oplus, \oplus') \in (a \rightarrow b \rightarrow b)$ is equivalent to

$$\begin{array}{l} \text{for all } x \in A, x' \in A', y \in B, y' \in B', \\ \quad a\ x = x' \text{ and } b\ y = y' \text{ implies } b\ (x \oplus y) = x' \oplus' y'. \end{array}$$

The result as a whole may then be rephrased,

$$\begin{array}{l} \text{for all } a : A \rightarrow A', b : B \rightarrow B', \\ \quad \text{if for all } x \in A, y \in B, \quad b\ (x \oplus y) = (a\ x) \oplus' (b\ y), \\ \quad \text{and } b\ u = u' \\ \quad \text{then } b \circ \mathit{fold}_{AB}\ (\oplus)\ u = \mathit{fold}_{A'B'}\ (\oplus')\ u' \circ a^*. \end{array}$$

The theorems derived from types can often be given a reading with an algebraic flavour, and the result about *fold* provides an illustration of this. Let $(A, B, \oplus, u)$ and $(A', B', \oplus', u')$ be two algebraic structures. The functions a and b form a homomorphism between these if $b\ (x \oplus y) = (a\ x) \oplus' (b\ y)$ for all x and y, and if $b\ u = u'$. Similarly, let $(A^*, B, \mathit{fold}_{AB}\ (\oplus)\ u)$ and $(A'^*, B', \mathit{fold}_{A'B'}\ (\oplus')\ u')$ also be two algebraic structures. The functions a^* and b form a homomorphism between these if $b\ (\mathit{fold}_{AB}\ (\oplus)\ u\ xs) = \mathit{fold}_{A'B'}\ (\oplus')\ u'\ (a^*\ xs)$. The result about *fold* states that if a and b form a homomorphism between (A, B, c, n) and (A', B', c', n'), then a^* and b form a homomorphism between $(A^*, B, \mathit{fold}_{AB}\ (\oplus)\ u)$ and $(A'^*, B', \mathit{fold}_{A'B'}\ (\oplus')\ u')$.

3.3 Sorting

Let s be a closed term of the type

$$s : \forall X.(X \rightarrow X \rightarrow \mathit{Bool}) \rightarrow (X^* \rightarrow X^*)$$

Functions of this type include *sort* and *nub*:

$$\begin{array}{rcl} \mathit{sort}_{\mathit{Int}}(<_{\mathit{Int}})[3,1,4,2,5] & = & [1,2,3,4,5] \\ \mathit{nub}_{\mathit{Int}}(=_{\mathit{Int}})[1,1,2,2,2,1] & = & [1,2,1] \end{array}$$

The function *sort* takes an ordering function and a list and returns the list sorted in ascending order, and the function *nub* takes an equality predicate and a list and returns the list with adjacent duplicates removed.

Applying parametricity to the type of s yields, for all $a : A \rightarrow A'$,

$$\begin{array}{c} \text{if for all } x, y \in A, \quad (x \prec y) = (a\ x \prec' a\ y) \quad \text{then} \\ a^* \circ s_A(\prec) = s_{A'}(\prec') \circ a^* \end{array}$$

(Recall that *Bool* as a relation is just the identity relation of booleans.) As a corollary, we have

$$\begin{array}{c} \text{if for all } x, y \in A, \quad (x < y) = (a\ x <' a\ y) \quad \text{then} \\ \mathit{sort}_{A'}\ (<) \circ a^* = a^* \circ \mathit{sort}_A\ (<') \end{array}$$

so maps commute with *sort*, when the function mapped preserves ordering. (If $<$ and $<'$ are linear orderings, then the hypothesis is equivalent to requiring that a is monotonic.) As a second corollary, we have

$$\begin{array}{c} \text{if for all } x, y \in A, \quad (x \equiv y) = (a\ x \equiv' a\ y) \quad \text{then} \\ \mathit{nub}_{A'}\ (\equiv) \circ a^* = a^* \circ \mathit{nub}_A\ (\equiv') \end{array}$$

so maps commute with *nub*, when the function mapped preserves equivalence. (If $\equiv$ and $\equiv'$ are equality on A and A', then the hypothesis is equivalent to requiring that a is one-to-one.)

3.4 Polymorphic equality

The programming language Miranda [Tur85] provides a polymorphic equality function, with type

$$(=) : \forall X.\ X \rightarrow X \rightarrow \mathit{Bool}.$$

Applying parametricity to the type of $(=)$ yields, for all $a : A \rightarrow A'$,

$$\text{for all } x, y \in A, \quad (x =_A y) = (a\ x =_{A'} a\ y).$$

This is obviously false; it does not hold for all a, but only for functions a that are one-to-one.

This is not a contradiction to the parametricity theorem; rather, it provides a proof that polymorphic equality cannot be defined in the pure polymorphic lambda calculus. Polymorphic equality can be added as a constant, but then parametricity will not hold (for terms containing the constant).

This suggests that we need some way to "tame" the power of the polymorphic equality operator. Exactly such taming is provided by the "eqtype variables" of Standard ML [Mil87], or more generally by the "type classes" of Haskell [HW88,WB89]. In these languages, we can think of polymorphic equality as having the type

$$(=) : \forall^{(=)}X.\ X \rightarrow X \rightarrow \mathit{Bool}.$$

Here $\forall^{(=)}X.\ F(X)$ is a new type former, where X ranges only over types for which equality is defined. Corresponding to the type constructor $\forall^{=}$ is a new relation constructor:

$$\begin{array}{c} (g, g') \in \forall^{(=)}\mathcal{X}.\ \mathcal{F}(\mathcal{X}) \\ \text{iff} \\ \text{for all } \mathcal{A} : A \Leftrightarrow A' \text{ respecting } (=), \quad (g_A, g'_{A'}) \in \mathcal{F}(\mathcal{A}). \end{array}$$

A relation $\mathcal{A} : A \Leftrightarrow A'$ respects $(=)$ if whenever $x =_A y$ and $(x, x') \in \mathcal{A}$ and $(y, y') \in \mathcal{A}$ then $x' =_{A'} y'$, where $(=_A)$ is equality on A and $(=_{A'})$ is equality on A'. In the case where $\mathcal{A}$ is a function a, this is equivalent to requiring that a be one-to-one.

With this definition, we can prove that the polymorphic equality operator, typed as above, satisfies the parametricity theorem. In our extended language we can define, for example, the function

$$nub : \forall^{(=)}X.\ X^* \to X^*$$

and the corresponding parametricity condition is the same as that for the previous version of *nub*.

Thus, the more refined type structures of Standard ML and Haskell add exactly the information necessary to maintain parametricity. In Standard ML this trick works only for equality (which is built into the language), whereas in Haskell it works for any operators defined using the type class mechanism.

3.5 A result about map

Suppose that I tell you that I am thinking of a function m with the type

$$m : \forall X.\forall Y.(X \to Y) \to (X^* \to Y^*)$$

You will immediately guess that I am thinking of the map function, $m(f) = f^*$. Of course, I could be thinking of a different function, for instance, one that reverses a list and then applies f^* to it. But intuitively, you know that map is the only interesting function of this type: that all others must be rearranging functions composed with map.

We can formalise this intuition as follows. Let m be a function with the type above. Then

$$m_{AB}(f) = f^* \circ m_{AA}(I_A) = m_{BB}(I_B) \circ f^*$$

where I_A is the identity function on A. The function $m_{AA}(I_A)$ is a rearranging function, as discussed in the preceding section. Thus, every function m of the above type can be expressed as a rearranging function composed with map, or equivalently, as map composed with a rearranging function.

The proof is simple. As we have already seen, the parametricity condition for m is that

$$\text{if } f' \circ a = b \circ f \text{ then } m_{A'B'}(f') \circ a^* = b^* \circ m_{AB}(f)$$

Taking $A' = B' = B$, $b = f' = I_B$, $a = f$ satisfies the hypotheses, giving as the conclusion

$$m_{BB}(I_B) \circ f^* = (I_B)^* \circ m_{AB}(f)$$

which gives us the second equality above, since $(I_B)^* = I_{B^*}$. The first equality may be derived by commuting the permuting function with map; or may be derived directly by a different substitution.

3.6 A result about fold

Analogous to the previous result about *map* is a similar result about *fold*. Let f be a function with the type

$$f : \forall X.\forall Y.(X \to Y \to Y) \to Y \to X^* \to Y$$

Then

$$f_{AB}\ c\ n = fold_{AB}\ c\ n \circ f_{AA^*}\ cons_A\ nil_A$$

Note that $f_{AA^*}\ cons_A\ nil_A : A^* \to A^*$ is a function that rearranges a list, so this says that every function with the type of *fold* can be expressed as *fold* composed with a rearranging function.

The proof is similar to the previous one. The parametricity condition for f is that

$$\begin{gathered}\text{if } c' \circ (a \times b) = b \circ c \text{ and } n' = b(n) \text{ then}\\ f_{A'B'}\ c'\ n' \circ a^* = b \circ f_{AB}\ c\ n\end{gathered}$$

Taking $A = A'$, $B = A^*$, $a = I_A$, $b = fold_{A'B'}\ c'\ n'$, $c = cons_A$, $n = nil_A$ satisfies the hypothesis, giving as the conclusion

$$f_{AB'}\ c'\ n' \circ I_A^* = fold_{AB'}\ c'\ n' \circ f_{AA^*}\ cons_A\ nil_A$$

The I_A^* term is just an identity, and so drops out, leaving us with the desired equality if we rename c', n', B' to c, n, B.

3.7 A result about filter

Let f be a function with the type

$$f : \forall X.(X \to Bool) \to X^* \to X^*$$

Three functions with this type are *filter*, *takewhile*, and *dropwhile*. For example,

$$\begin{aligned} filter\ odd\ [3,1,4,5,2] &= [3,1,5]\\ takewhile\ odd\ [3,1,4,5,2] &= [3,1]\\ dropwhile\ odd\ [3,1,4,5,2] &= [4,5,2]\end{aligned}$$

See [BW88] for the definitions of these functions.

For every such f we can define a corresponding function of type

$$g : \forall X.(X \times Bool)^* \to X^*$$

such that f and g are related by the equation

$$f_A(p) = g_A \circ \langle I_A, p\rangle \qquad (*)$$

where $\langle I_A, p\rangle\ x = (x, p\ x)$. That is, f_A is passed a predicate p of type $A \to Bool$ and a list of A, whereas g_A is passed a list of $A \times Bool$ pairs, the second component of the pair being the result of applying p to the first component. Intuitively, this transformation is possible because the only values that p can be applied to are of type A, so it suffices to pair each value of type A with the result of applying p to it.

A little thought shows that a suitable definition of g is

$$g_A = fst^* \circ f_{A\times Bool}(snd)$$

We can use parametricity to show that f and g satisfy $(*)$, for all functions f of the given type. The parametricity conditions for f tells us that for any $a : A \rightarrow A'$ and any $p : A \rightarrow Bool$ and $p' : A' \rightarrow Bool$ we have

$$\text{if } p' \circ a = I_{Bool} \circ p \text{ then } f_{A'}(p') \circ a^* = a^* \circ f_A(p)$$

Take $A' = A \times Bool$ and $a = \langle I_A, p\rangle$ and $p' = snd$. Then the hypothesis becomes $snd \circ \langle I_A, p\rangle = p$, which is satisfied, yielding the conclusion

$$f_{A\times Bool}(snd) \circ \langle I_A, p\rangle^* = \langle I_A, p\rangle \circ f_A(p).$$

Compose both sides with fst^*, giving

$$fst^* \circ f_{A\times Bool}(snd) \circ \langle I_A, p\rangle^* = fst^* \circ \langle I_A, p\rangle \circ f_A(p).$$

Then apply the definition of g, and observe that $fst \circ \langle I_A, p\rangle = I_A$, resulting in the equation

$$g_A \circ \langle I_A, p\rangle^* = f_A(p)$$

as desired.

3.8 An isomorphism

The preceding applications can all be expressed in the Hindley/Milner fragment of the polymorphic lambda calculus: all universal quantifiers appear at the outside of a type. This section presents an application that utilises the full power of the Girard/Reynolds system.

Let A be an arbitrary type. Intuitively, this type is isomorphic to the type $\forall X.\ (A \rightarrow X) \rightarrow X$, which we will abbreviate as $\tilde{A}$. The apparent isomorphism between A and $\tilde{A}$ is expressed by the functions:

$$\begin{aligned} i &: A \rightarrow \tilde{A} \\ i &= \lambda x : A.\ \Lambda X.\ \lambda g : A \rightarrow X.\ g\ x \\ j &: \tilde{A} \rightarrow A \\ j &= \lambda h : \tilde{A}.\ h_A\ (\lambda x : A.\ x) \end{aligned}$$

That is, i takes an element x of A to the element of $\tilde{A}$ that maps a function g (of type $A \rightarrow X$) to the value $g\ x$ (of type X). The inverse function j recovers the original element by applying a value in $\tilde{A}$ to the identity function.

To prove that this truly is an isomorphism, we must verify that $j \circ i$ and $i \circ j$ are both identities. It is easy enough to verify the former:

$$\begin{aligned} & j\ (i\ x) \\ = {} & j\ (\Lambda X.\ \lambda g : A \rightarrow X.\ g\ x) \\ = {} & (\lambda g : A \rightarrow A.\ g\ x)\ (\lambda x : A.\ x) \\ = {} & (\lambda x : A.\ x)\ x \\ = {} & x \end{aligned}$$

However, the inverse identity is problematic. We can get as far as

$$\begin{aligned} & i\ (j\ h) \\ = {} & i\ (h_A\ (\lambda x : A.\ x)) \\ = {} & \Lambda X.\ \lambda g : A \rightarrow X.\ g\ (h_A\ (\lambda x : A.\ x)) \end{aligned}$$

and now we are stuck. Here is where parametricity helps. The parametricity condition for $h : \forall X.\ (A \rightarrow X) \rightarrow X$ is that, for all $b : B \rightarrow B'$ and all $f : A \rightarrow B$,

$$b\ (h_B\ f) = h_{B'}\ (b \circ f)$$

Taking $B = A$, $B' = X$, $b = g$, and $f = (\lambda x : A.\ x)$ gives

$$\begin{aligned} & \Lambda X.\ \lambda g : A \rightarrow X.\ g\ (h_A\ (\lambda x : A.\ x)) \\ = {} & \Lambda X.\ \lambda g : A \rightarrow X.\ h_X\ (g \circ (\lambda x : A.\ x)) \\ = {} & \Lambda X.\ \lambda g : A \rightarrow X.\ h_X\ g \\ = {} & h \end{aligned}$$

which completes the second identity.

The second identity depends critically on parametricity, so the isomorphism holds only for models in which all elements satisfy the parametricity constraint. Alas, the parametricity theorem guarantees only that elements of the model that correspond to lambda terms will be parametric; many models contain additional elements that are non-parametric. One model that contains only parametric elements is that in [BTC88].

4 Polymorphic lambda calculus

We now turn to a more formal development of the parametricity theorem. We begin with a quick review of the polymorphic lambda calculus.

We will use X, Y, Z to range over type variables, and T, U, V to range over types. Types are formed from type variables, function types, and type abstraction:

$$T ::= X \mid T \rightarrow U \mid \Lambda X.\ T$$

We will use x, y, z to range over individual variables, and t, u, v to range over terms. Terms are formed from individual variables, abstraction and application of individuals, and abstraction and application of types:

$$t ::= x \mid \lambda x : U.\ t \mid t\ u \mid \Lambda X.\ t \mid t_U$$

$$\bar{X}; \bar{x}, x : T \vdash x : T$$

$$\to\mathcal{I}\,\frac{\bar{X}; \bar{x}, x : U \vdash v : V}{\bar{X}; \bar{x} \vdash \lambda x : U.\ v : U \to V}$$

$$\to\mathcal{E}\,\frac{\bar{X}; \bar{x} \vdash t : U \to V \quad \bar{X}; \bar{x} \vdash u : U}{\bar{X}; \bar{x} \vdash t\ u : V}$$

$$\forall\mathcal{I}\,\frac{\bar{X}; \bar{x} \vdash t : T}{\bar{X}, X; \bar{x} \vdash \Lambda X.\ t : \forall X.\ T}$$

$$\forall\mathcal{E}\,\frac{\bar{X}; \bar{x} \vdash t : \forall X.\ T}{\bar{X}; \bar{x} \vdash t_U : T[U/X]}$$

Figure 2: Typing rules

We write $T[U/X]$ to denote substitution of U for the free occurrences of X in T, and $t[u/x]$ and $t[U/X]$ similarly.

A term is legal only if it is well typed. Typings are expressed as assertions of the form

$$\bar{X}; \bar{x} \vdash t : T$$

where $\bar{X}$ is a list of distinct type variables $X_1, \ldots, X_m$, and $\bar{x}$ is a list of distinct individual variables, with types, $x_1 : T_1, \ldots, x_n : T_n$. This assertion may be read as stating that t has type T in a context where each x_i has type T_i. Each individual variable that appears free in t should appear in $\bar{x}$, and each type variable that appears free in T of $\bar{x}$ should appear in $\bar{X}$. The type inference rules are shown in Figure 2.

Two terms are equivalent if one can be derived from the other by renaming bound individual or type variables (α conversion). In addition, we have the familiar reduction rules:

$$\begin{array}{lrcl}
(\beta) & (\lambda x : U.\ t)\ u & \Rightarrow & t[u/x] \\
 & (\Lambda X.\ t)_U & \Rightarrow & t[U/X] \\
(\eta) & \lambda x : U.\ t\ x & \Rightarrow & t \\
 & \Lambda X.\ t_X & \Rightarrow & t
\end{array}$$

where in the η rules x and X do not occur free in t.

As is well known, familiar types such as booleans, pairs, lists, and natural numbers can be defined as types constructed from just $\to$ and $\forall$; see for example [Rey85] or [GLT89]. Alternatively, we could add suitable types and individual constants to the pure language described above.

5 Semantics of polymorphic lambda calculus

We will give a semantics using a version of the frame semantics outlined in [BM84] and [MM85]. We first discuss the semantics of types, and then discuss the semantics of terms.

5.1 Types

A type model consists of a universe $\mathbf{U}$ of type values, and two operations, $\to$ and $\forall$ that construct types from other types. There is a distinguished set $[\mathbf{U} \to \mathbf{U}]$ of functions from $\mathbf{U}$ to $\mathbf{U}$. If A and B are in $\mathbf{U}$, then $A \to B$ must be in $\mathbf{U}$, and if F is in $[\mathbf{U} \to \mathbf{U}]$, then $\forall F$ must be in $\mathbf{U}$.

Let T be a type with its free variables in $\bar{X}$. We say that $\bar{A}$ is a type environment for $\bar{X}$ if it maps each type variable in $\bar{X}$ into a type value in $\mathbf{U}$. The corresponding value of T in the environment $\bar{A}$ is written $[\![T]\!]\bar{A}$ and is defined as follows:

$$\begin{array}{lcl}
[\![X]\!]\bar{A} & = & \bar{A}[\![X]\!] \\
[\![T \to U]\!]\bar{A} & = & [\![T]\!]\bar{A} \to [\![U]\!]\bar{A} \\
[\![\forall X.\ T]\!]\bar{A} & = & \forall(\lambda A.\ [\![T]\!]\bar{A}[A/X])
\end{array}$$

Here $\bar{A}[\![X]\!]$ is the value that $\bar{A}$ maps X into, and $\bar{A}[A/X]$ is the environment that maps X into A and otherwise behaves as $\bar{A}$. (The reader may find that the above looks more familiar if $\bar{A}$ is replaced everywhere by a Greek letter such as η.)

5.2 Terms

Associated with each type A in $\mathbf{U}$ is a set $\mathbf{D}_A$ of the values of that type.

For each A and B in $\mathbf{U}$, the elements in $\mathbf{D}_{A\to B}$ represent functions from $\mathbf{D}_A$ to $\mathbf{D}_B$. We do not require that the elements *are* functions, merely that they represent functions. In particular, associated with each A and B in $\mathbf{U}$ there must be a set $[\mathbf{D}_A \to \mathbf{D}_B]$ of functions from $\mathbf{D}_A$ to $\mathbf{D}_B$, and functions

$$\begin{array}{lcl} \phi_{A,B} & : & \mathbf{D}_{A\to B} \to [\mathbf{D}_A \to \mathbf{D}_B] \\ \psi_{A,B} & : & [\mathbf{D}_A \to \mathbf{D}_B] \to \mathbf{D}_{A\to B} \end{array}$$

such that $\phi_{A,B} \circ \psi_{A,B}$ is the identity on $[\mathbf{D}_A \to \mathbf{D}_B]$. We will usually omit the subscripts and just write ϕ and ψ.

If F is a function in $[\mathbf{U} \to \mathbf{U}]$, the elements in $\mathbf{D}_{\forall F}$ represent functions that take a type A into an element of $\mathbf{D}_{F(A)}$. In particular, associated with each F there must be a set $[\forall A : \mathbf{U}.\ \mathbf{D}_{F(A)}]$ of functions that map each A in $\mathbf{U}$ into an element of $\mathbf{D}_{F(A)}$, and functions

$$\begin{array}{lcl} \Phi_F & : & \mathbf{D}_{\forall F} \to [\forall A : \mathbf{U}.\ \mathbf{D}_{F(A)}] \\ \Psi_F & : & [\forall A : \mathbf{U}.\ \mathbf{D}_{F(A)}] \to \mathbf{D}_{\forall F} \end{array}$$

such that $\Phi_F \circ \Psi_F$ is the identity on $[\forall A : \mathbf{U}.\ \mathbf{D}_{F(A)}]$. Again, we will usually omit the subscripts and just write Φ and Ψ.

Let t be a term such that $\bar{X}; \bar{x} \vdash t : T$. We say that $\bar{A}, \bar{a}$ are environments respecting $\bar{X}, \bar{x}$ if $\bar{A}$ is a type environment for $\bar{X}$ and $\bar{a}$ is an environment mapping variables to values such that for each $x_i : T_i$ in $\bar{x}$, we have that $\bar{a}[\![x_i]\!] \in \mathbf{D}_{[\![T_i]\!]\bar{A}}$. The value of t in the environments $\bar{A}$ and $\bar{a}$ is written $[\![t]\!]\bar{A}\,\bar{a}$ and is defined as follows:

$$\boxed{\begin{array}{lcl} [\![x]\!]\bar{A}\,\bar{a} & = & \bar{a}[\![x]\!] \\ [\![\lambda x : U.\ v]\!]\bar{A}\,\bar{a} & = & \psi\ (\lambda a.\ [\![v]\!]\bar{A}\,\bar{a}[a/x]) \\ [\![t\,u]\!]\bar{A}\,\bar{a} & = & \phi\ ([\![t]\!]\bar{A}\,\bar{a})\ ([\![u]\!]\bar{A}\,\bar{a}) \\ [\![\Lambda X.\ v]\!]\bar{A}\,\bar{a} & = & \Psi\ (\lambda A.\ [\![v]\!]\bar{A}[A/X]\,\bar{a}) \\ [\![t_U]\!]\bar{A}\,\bar{a} & = & \Phi\ ([\![t]\!]\bar{A}\,\bar{a})\ ([\![U]\!]\bar{A}) \end{array}}$$

Here $\bar{a}[\![x]\!]$ is the value that $\bar{a}$ maps x into, and $\bar{a}[a/x]$ is the environment that maps x into a and otherwise behaves as $\bar{a}$.

A *frame* is a structure specifying $\mathbf{U}, \to, \forall$ and $\mathbf{D}, \phi, \psi, \Phi, \Psi$ satisfying the constraints above. A frame is an *environment model* if for every $\bar{X}; \bar{x} \vdash t : T$ and every $\bar{A}, \bar{a}$ respecting $\bar{X}, \bar{x}$, the meaning of $[\![t]\!]\bar{A}\,\bar{a}$ as given above exists. (That is, a frame is a model if the sets $[\mathbf{U} \to \mathbf{U}]$, $[[\mathbf{D}_A \to \mathbf{D}_B]$, and $[\forall A : \mathbf{U}.\ \mathbf{D}_{F(A)}]$ are "big enough".)

We write $\bar{X}; \bar{x} \models t : T$ if for all environments $\bar{A}, \bar{a}$ respecting $\bar{X}, \bar{x}$, we have $[\![t]\!]\bar{A}\,\bar{a} \in \mathbf{D}_{[\![T]\!]\bar{A}}$.

> **Proposition.** *(Soundness of types.) For all $\bar{X}, \bar{x}, t$ and T, if $\bar{X}; \bar{x} \vdash t : T$ then $\bar{X}; \bar{x} \models t : T$.*

The type soundness result simply states that the meaning of a typed term corresponds to the meaning of the corresponding type. The proof is a straightforward induction over the structure of type inferences. Parametricity is an analogue of this result, as we shall see in the next section.

6 The parametricity theorem

In the previous section, we defined a semantics where a type environment $\bar{A}$ consists of a mapping of type variables onto types, and the semantics of a type T in the environment $\bar{A}$ is a set denoted $\mathbf{D}_{[\![T]\!]\bar{A}}$. In this section, we define an alternative semantics where a type environment $\bar{\mathcal{A}}$ consists of a mapping of type variables onto relations, and the semantics of a type T in the environment $\bar{\mathcal{A}}$ is a relation denoted $[\![T]\!]\bar{\mathcal{A}}$.

We can then formally state the parametricity theorem: terms in related environments have related values. We can think of environments $\bar{A}$ and $\bar{A}'$ as specifying two different representations of types, related by $\bar{\mathcal{A}}$, which is why Reynolds' called his version of this result "the abstraction theorem". A key point of this paper is that this theorem has applications other than change of representation, hence the change in name from "abstraction" to "parametricity"

A function type may be regarded as a relation as follows. If $\mathcal{A} : A \Leftrightarrow A'$ and $\mathcal{B} : B \Leftrightarrow B'$ are two relations, then we define

$$\mathcal{A} \to \mathcal{B} : (A \to B) \Leftrightarrow (A' \to B')$$

to be the relation

$$\mathcal{A} \to \mathcal{B} = \{\ (f, f')\ \mid\ \begin{array}{l}(a, a') \in \mathcal{A} \text{ implies} \\ (\phi\, f\, a, \phi\, f'\, a') \in \mathcal{B}\ \}\end{array}$$

In other words, functions are related if they map related arguments into related results.

A type abstraction may be regarded as a relation as follows. Let F be a function from $\mathbf{U}$ to $\mathbf{U}$, and F' be a function from $\mathbf{U}'$ to $\mathbf{U}'$, and for each A in $\mathbf{U}$ and A' in $\mathbf{U}'$, let $\mathcal{F}$ be a function that takes a relation $\mathcal{A} : A \Leftrightarrow A'$ and returns a relation $\mathcal{F}(\mathcal{A}) : F(A) \Leftrightarrow F'(A')$. Then we define

$$\forall\mathcal{F} : \forall F \Leftrightarrow \forall F'$$

to be the relation

$$\forall\mathcal{F} = \{\ (g, g')\ |\ \text{for all } A,\ A',\ \text{and } \mathcal{A} : A \Leftrightarrow A',\ (\Phi(g)(A), \Phi(g')(A')) \in \mathcal{F}(\mathcal{A})\ \}$$

In other words, type abstractions are related if they map related types into related results.

A relation environment maps each type variable into a relation. Let $\bar{\mathcal{A}}$ be a relation environment for $\bar{X}$, and let $\bar{A}, \bar{A}'$ be two type environments for $\bar{X}$. We write $\bar{\mathcal{A}} : \bar{A} \Leftrightarrow \bar{A}'$ if for each X in $\bar{X}$ we have $\bar{\mathcal{A}}[\![X]\!] : \bar{A}[\![X]\!] \Leftrightarrow \bar{A}'[\![X]\!]$.

Given a relation environment $\bar{\mathcal{A}}$ we can interpret a type T as a relation $[\![T]\!]\bar{\mathcal{A}}$ as follows:

$$\boxed{\begin{array}{lcl} [\![X]\!]\bar{\mathcal{A}} & = & \bar{\mathcal{A}}[\![X]\!] \\ [\![U \rightarrow V]\!]\bar{\mathcal{A}} & = & [\![U]\!]\bar{\mathcal{A}} \rightarrow [\![V]\!]\bar{\mathcal{A}} \\ [\![\forall X.\ V]\!]\bar{\mathcal{A}} & = & \forall(\lambda\mathcal{A}.\ [\![V]\!]\bar{\mathcal{A}}[\mathcal{A}/X]) \end{array}}$$

Let $\bar{A}, \bar{a}$ respect $\bar{X}, \bar{x}$ and $\bar{A}', \bar{a}'$ respect $\bar{X}, \bar{x}$. We say that $\bar{\mathcal{A}}, \bar{A}, \bar{A}', \bar{a}, \bar{a}'$ respect $\bar{X}, \bar{x}$ if $\bar{\mathcal{A}} : \bar{A} \Leftrightarrow \bar{A}'$ and $(\bar{a}[\![x_i]\!], \bar{a}'[\![x_i]\!]) \in [\![T_i]\!]\bar{\mathcal{A}}$ for each $x_i : T_i$ in $\bar{x}$. It is easy to see that if $\bar{\mathcal{A}}, \bar{A}, \bar{A}', \bar{a}, \bar{a}'$ respect $\bar{X}, \bar{x}$ then $\bar{A}, \bar{a}$ respect $\bar{X}, \bar{x}$ and $\bar{A}', \bar{a}'$ respect $\bar{X}, \bar{x}$.

We say that $\bar{X}; \bar{x} \models t : T$ iff for every $\bar{\mathcal{A}}, \bar{A}, \bar{A}', \bar{a}, \bar{a}'$ that respect $\bar{X}, \bar{x}$ we have $([\![t]\!]\bar{A}\bar{a}, [\![t]\!]\bar{A}'\bar{a}') \in [\![T]\!]\bar{\mathcal{A}}$.

Proposition. *(Parametricity.) For all $\bar{X}$, $\bar{x}$, t, and T, if $\bar{X}; \bar{x} \vdash t : T$ then $\bar{X}; \bar{x} \models t : T$.*

Proof. The proof is a straightforward induction over the structure of type inferences. For each of the inference rules in Figure 2, we replace $\vdash$ by $\models$ and show that the resulting inference is valid. (End of proof.)

As mentioned previously, data types such as booleans, pairs, lists, and natural numbers can be defined in terms of $\rightarrow$ and $\forall$.

As an example, consider the construction for pairs. The type $X \times Y$ is defined as an abbreviation:

$$X \times Y \stackrel{\text{def}}{=} \forall Z.\ X \rightarrow Y \rightarrow Z$$

Every term of type $X \times Y$ is equivalent to a term of the form $pair_{XY}\ x\ y$, where $x : X$ and $y : Y$, and $pair$ is defined by

$$\begin{array}{rl} pair \stackrel{\text{def}}{=} & \Lambda X.\ \Lambda Y.\ \lambda x : X.\ \lambda y : Y. \\ & \Lambda Z.\ \lambda p : X \rightarrow Y \rightarrow Z.\ p\ x\ y \end{array}$$

The type of $pair$ is, of course,

$$pair : \forall X.\ \forall Y.\ X \rightarrow Y \rightarrow X \times Y$$

where $X \times Y$ stands for the abbreviation above. It follows from the parametricity theorem that if $\mathcal{A} : A \rightarrow A'$ and $\mathcal{B} : B \rightarrow B'$, and $(a, a') \in \mathcal{A}$ and $(b, b') \in \mathcal{B}$, then

$$\begin{array}{l} (\ [\![pair_{XY}\ x\ y]\!][A/X, B/Y]\,[a/x, b/y], \\ \ \ [\![pair_{XY}\ x\ y]\!][A'/X, B'/Y]\,[a'/x, b'/y]\ ) \\ \in [\![X \times Y]\!][\mathcal{A}/X, \mathcal{B}/Y]. \end{array}$$

That is, pairs are related if their corresponding components are related, as we would expect.

It can be shown similarly, using the standard construction for lists, that lists are related if they have the same length and corresponding elements are related.

Alternatively, suitable type constructors and individual constants may be added to the pure polymorphic lambda calculus. In this case, for each new type constructor an appropriate corresponding relation must be defined; suitable definitions of relations for pair and list types were given in Section 2. Further, for each new constant the parametricity condition must be verified: if c is a constant of type T, we must check that $\models c : T$ holds. It then follows that parametricity holds for any terms built from the new type constructors and constants.

7 Fixpoints

Every term in typed lambda calculus is strongly normalising, so if a fixpoint operator is desired it must be added as a primitive. This section mentions the additional requirements necessary to ensure that the fixpoint primitive satisfies the abstraction theorem.

Frame models associate with each type A a set $\mathbf{D}_A$. In order to discuss fixpoints, we require that each set have sufficient additional structure to be a domain: it must be provided with an ordering $\sqsubseteq$ such that each domain has a least element, $\bot$, and such that limits of directed sets exist. Obviously, we also require that all functions are continuous.

What are the requirements on relations? The obvious requirement is that they, too, be continuous. That is, if $\mathcal{A} : A \Leftrightarrow A'$, and x_i is a chain in A, and x'_i is a chain in A', and $(x_i, x'_i) \in \mathcal{A}$ for every i, then we require that $(\bigsqcup x_i, \bigsqcup x'_i) \in \mathcal{A}$ also. But in addition to this, we need a second requirement, namely that each relation $\mathcal{A}$ is strict, that is, that $(\bot_A, \bot_{A'}) \in \mathcal{A}$. If we restrict relations in this way, then it is no longer true that every function $a : A \rightarrow A'$ may be treated as a relation; only strict functions may be treated as such.

With this restricted view of relations, it is easy to show that the fixpoint operator satisfies the parametricity theorem. As usual, for each type A define fix_A as the function

$$fix : \forall X.\, (X \rightarrow X) \rightarrow X$$

such that $fix_A\ f = \bigsqcup f^i\ \bot_A$. Parametricity holds if $(fix, fix) \in \forall \mathcal{A}.\ (\mathcal{A} \rightarrow \mathcal{A}) \rightarrow \mathcal{A}$. This will be true if for each $\mathcal{A} : A \Leftrightarrow A'$ and each $(f, f') \in \mathcal{A} \rightarrow \mathcal{A}$ we have $(fix_A\ f, fix_{A'}\ f') \in \mathcal{A}$. Recall that the condition on f and f' means that if $(x, x') \in \mathcal{A}$ then $(f\ x, f'\ x') \in \mathcal{A}$. Now, since all relations are strict, it follows that $(\bot_A, \bot_{A'}) \in \mathcal{A}$; hence $(f\ \bot_A, f'\ \bot_{A'}) \in \mathcal{A}$; and, in general, $(f^i\ \bot_A, f'^i\ \bot_{A'}) \in \mathcal{A}$. It follows, since all relations are continuous, that $(\bigsqcup f^i\ \bot_A, \bigsqcup f'^i\ \bot_{A'}) \in \mathcal{A}$, as required.

Note that the restriction to strict relations here is similar to the restriction to strict coercion functions in [BCGS89], and is adopted for similar reasons.

The requirement that relations are strict is essential. For a counterexample, take A to be the domain $\{\bot, true, false\}$, and take $\mathcal{A} : A \rightarrow A$ to be the constant relation such that $(x, true) \in \mathcal{A}$ for all x. The relation $\mathcal{A}$ is continuous but not strict. Let f be the constant function $f\ x = false$ and let f' the identity function $f'\ x = x$. Then $\mathcal{A} \rightarrow \mathcal{A}$ relates f to f', but $\mathcal{A}$ does not relate $fix_A\ f = false$ to $fix_A\ f' = \bot$.

The restriction to strict arrows is not to be taken lightly. For instance, given a function r of type

$$r : \forall A. A^* \rightarrow A^*$$

parametricity implies that

$$r_{A'} \circ a^* = a^* \circ r_A$$

for all functions $a : A \rightarrow A'$. If the fixpoint combinator appears in the definition of r, then we can only conclude that the above holds for strict a, which is a significant restriction.

The desire to derive theorems from types therefore suggests that it would be valuable to explore programming languages that prohibit recursion, or allow only its restricted use. In theory, this is well understood; we have already noted that any computable function that is provably total in second-order Peano arithmetic can be defined in the pure polymorphic lambda calculus, without using the fixpoint as a primitive. However, practical languages based on this notion remain *terra incognita*.

References

[BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov, Inheritance and explicit coercion. In *4'th Annual Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

[BFSS87] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott, Functorial polymorphism. In G. Huet, editor, *Logical Foundations of Functional Programming*, Austin, Texas, 1987. Addison-Wesley, to appear.

[BM84] K. B. Bruce and A. R. Meyer, The semantics of second-order polymorphic lambda calculus. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, Sophia-Antipolis, France, 1984, pp. 131–144. LNCS 173, Springer-Verlag.

[BTC88] V. Breazu-Tannen and T. Coquand, Extensional models for polymorphism. *Theoretical Computer Science*, 59:85–114, 1988.

[BW86] G. Barrett and P. Wadler, Derivation of a pattern-matching compiler. Manuscript, Programming Research Group, Oxford, 1986.

[BW88] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall, 1988.

[DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings of the 9'th Annual Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.

[deB89] P. J. deBruin, Naturalness of polymorphism. Submitted to *Category Theory and Computer Science*, Manchester, 1989.

[FGSS88] P. J. Freyd, J. Y. Girard, A. Scedrov, and P. J. Scott, Semantic parametricity in polymorphic lambda calculus. In *3'rd Annual Symposium on Logic in Computer Science*, Edinburgh, Scotland, June 1988.

[FLO83] S. Fortune, D. Leivant, and M. O'Donnell, The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30(1):151–185, January 1983.

[Gir72] J.-Y. Girard, *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* Ph.D. thesis, Université Paris VII, 1972.

[Gir86] J.-Y. Girard, The system F of variable types, fifteen years later. *Theoretical Computer Science*, **45**, pp. 159–192.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types.* Cambridge University Press, 1989.

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc. 146*, pp. 29–60, December 1969.

[HW88] P. Hudak and P. Wadler, editors, *Report on the Functional Programming Language Haskell.* Technical Report YALEU/DCS/RR656, Yale University, Department of Computer Science, December 1988; also Technical Report, Glasgow University, Department of Computer Science, December 1988.

[Mes89] J. Meseguer, Relating models of polymorphism. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[Mil78] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, pp. 348–375, 1978.

[Mil84] R. Milner, A proposal for Standard ML. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Mil87] R. Milner, Changes to the Standard ML core language. Report ECS-LFCS-87-33, Edinburgh University, Computer Science Dept., 1987.

[Mit86] J. C. Mitchell, Representation independence and data abstraction. In *13'th ACM Symposium on Principles of Programming Languages*, pp. 263–276.

[MM85] J. C. Mitchell and A. R. Meyer, Second-order logical relations. In R. Parikh, editor, *Logics of Programs*, Brooklyn, New York, 1985. LNCS 193, Springer-Verlag.

[Pit87] A. M. Pitts, Polymorphism is set theoretic, constructively. In D. H. Pitt, *et al.*, editors, *Category Theory and Computer Science*, Edinburgh, 1987. LNCS 283, Springer-Verlag.

[Rey74] J. C. Reynolds, Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19, Springer-Verlag.

[Rey83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pp. 513–523. North-Holland, Amsterdam.

[Rey84] J. C. Reynolds, Polymorphism is not set theoretic. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, Sophia-Antipolis, France, 1984, pp. 145–156. LNCS 173, Springer-Verlag.

[Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.

[She89] M. Sheeran, Categories for the working hardware designer. In *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, Cornell, July 1989.

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.

[WB89] P. Wadler and S. Blott, How to make *ad-hoc* polymorphism less *ad hoc*. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.