Testing Polymorphic Properties

Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen

Chalmers University of Technology
{bernardy,patrikj,koen}@chalmers.se

Abstract. This paper is concerned with testing properties of polymorphic functions. The problem is that testing can only be performed on specific monomorphic instances, whereas parametrically polymorphic functions are expected to work for any type. We present a schema for constructing a monomorphic instance for a polymorphic property, such that correctness of that single instance implies correctness for all other instances. We also give a formal definition of the class of polymorphic properties the schema can be used for. Compared to the standard method of testing such properties, our schema leads to a significant reduction of necessary test cases.

Key words: polymorphism, parametricity, initiality, testing

1 Introduction

How should one test a polymorphic function?

A modern and convenient approach to testing is to write specifications as properties, and let a tool generate test cases. Such tools have been implemented for many programming languages, such as Ada, C++, Curry, Erlang, Haskell, Java, .NET and Scala [2, 3, 6, 7, 16, 20, 24, 27]. But how should one generate test cases for polymorphic functions? Parametrically polymorphic functions, by their very nature, work uniformly on values of any type, whereas in order to run a concrete test, one must pick values from a specific monomorphic type.

As an example, suppose we have two different implementations of the standard function *reverse* that reverses a list:

$$\mathit{reverse1}, \mathit{reverse2} : \forall a.\ [a] \to [a]$$

In order to test that they do the same thing, what monomorphic type should we pick for the type variable a? Standard praxis, as for example used by QuickCheck [7], suggests to simply use a type with a large enough domain, such as natural numbers, resulting in the following property:

$$\forall\ \mathit{xs} : [\mathbb{N}].\ \mathit{reverse1}\ \mathit{xs} == \mathit{reverse2}\ \mathit{xs}$$

Intuitively, testing the functions only on the type $\mathbb{N}$ is "enough"; if the original polymorphic property has a counter example (in this case a monomorphic type T and a concrete list $\mathit{xs} : [T]$), there also exists a counter example to the monomorphic property (in this case a concrete list $\mathit{xs}' : [\mathbb{N}]$).

However, how do we *know* this is enough? And, can we do better than this? This paper aims to provide an answer to these questions for a large class of

properties of polymorphic functions. We give a systematic way of computing the monomorphic type that a polymorphic property should be tested on. Perhaps surprisingly, we do this by only inspecting the type of the functions that are being tested, not their definition. Moreover, our method significantly improves on the standard testing praxis by making the monomorphic domains over which we quantify even more precise. For example, to check that *reverse1* and *reverse2* implement the same function, it turns out to be enough to test:

$$\forall\, n : \mathbb{N}.\ \mathit{reverse1}\ [1\,..\,n] == \mathit{reverse2}\ [1\,..\,n]$$

In other words, we only need to quantify over the *length* of the argument list, and not its elements! This is a big improvement over the previous property; for each list length n, only *one* test suffices, whereas previously, we had an unbounded number of lists to test for each length. This significantly increases test efficiency.

Related Work There are a few cases in the literature where it has been shown that, for a specific polymorphic function, testing it on a particular monomorphic type is enough. For example, Knuth's classical result that verifying a sorting network only has to be done on booleans [19, sec. 5.3.4], can be cast into a question about polymorphic testing [11]. The network can be represented as a polymorphic function parametrised over a comparator (a 2-element sorter):

$$\mathit{sort} : \forall a.\ (a \times a \to a \times a) \to [a] \to [a]$$

Knuth has shown that, in order to check whether such a function really sorts, it is enough to show that it works for booleans; in other words checking if the following function is a sorting function:

$$\mathit{sort\_Bool} : [\mathit{Bool}] \to [\mathit{Bool}]$$
$$\mathit{sort\_Bool} = \mathit{sort}\,(\lambda(x, y) \to (x \wedge y, x \vee y))$$

Another example is a result by Voigtländer [28], which says that in order to check that a given function is a scan function, it is enough to check it for all possible combinations on a domain of three elements.

The result we present in this paper has the same motivation as these earlier results, but the concrete details are not exactly the same. In section 4, we compare our general result with Knuth's and Voigtländer's specific results.

Contributions and outlook Our main contribution is a schema for testing polymorphic properties effectively and efficiently. We explain the schema both from a theoretical and practical point of view. Our examples are aimed at giving practitioners a good intuition of the method (section 2) and demonstrate some of its applications (section 4). A more formal exposition is provided in section 3. We cover related and future work in sections 5 and 6 and we conclude in section 7.

2 Examples

In this section, we discuss a number of examples illustrating the idea behind our method in preparation for the more formal treatment in the next section. We are using Haskell-like notation and QuickCheck-like properties here, but our result can be used in the context of other languages and other property-testing frameworks.

Example 1. Let us first compare two implementations of the function *filter*:

$$\mathit{filter1}, \mathit{filter2} : \forall a.\ (a \to \mathit{Bool}) \to [\,a\,] \to [\,a\,]$$

A parametric polymorphic function knows nothing about the type it is being used on. So, the only way an element of type a can appear in the result, is if it was produced somehow by the argument of the function. We can analyse the type of the arguments of the functions under test, in order to see in what way the arguments can be used to produce an element of type a. The monomorphic type A we are going to construct to test the functions on will represent all such ways in which the arguments can be used to produce an a.

In the case of *filter*, the only way we can produce elements of type a, is by using an element from its argument list (the predicate $(a \to \mathit{Bool})$ can only inspect elements). So, a natural choice for A is to be the index of the element from the argument list it used:

$$\mathbf{data}\ A = X\ \mathbb{N}$$

In other words, $X\ i$ stands for the i^{th} element (of type a) from the input list. Now, we have not only fixed a type to use for a, but also decided which elements the list xs should be filled with, once we know the length. Thus, the final monomorphic property becomes:

$$\begin{aligned}\forall\, n : \mathbb{N}, p : A \to \mathit{Bool}.\ \mathbf{let}\ & \mathit{xs} = [\,X\ 1\,..\,X\ n\,] \\ \mathbf{in}\ & \mathit{filter1}\ p\ \mathit{xs} == \mathit{filter2}\ p\ \mathit{xs}\end{aligned}$$

The construction we apply here can be seen as a kind of *symbolic simulation*: we feed the function with symbolic variables (here represented by naturals), and examine the output. This becomes more clear in the next example.

Example 2. Let us take a look at a typical polymorphic property, relating the functions *reverse* and append $(+\!\!+)$

$$\forall\, a{:}\,\star, \forall \mathit{xs}, \mathit{ys} : [\,a\,].\ \mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) == \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{reverse}\ \mathit{xs}$$

We can view the left- and right-hand sides of the property as two different polymorphic functions that are supposed to deliver the same result. Where can elements in the result list come from? Either from the list xs, or the list ys. Thus, the monomorphic type A becomes:

$$\mathbf{data}\ A = X\ \mathbb{N} \mid Y\ \mathbb{N}$$

And in the property, we not only instantiate the type, but also the elements of the lists:

$$\begin{aligned}\forall\, n, m : \mathbb{N}.\ \mathbf{let}\ & \mathit{xs} = [\,X\ 1\,..\,X\ n\,] \\ & \mathit{ys} = [\,Y\ 1\,..\,Y\ m\,] \\ \mathbf{in}\ & \mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) == \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{reverse}\ \mathit{xs}\end{aligned}$$

Finally, an example of a property that does not hold.

Example 3. Take a look at the following property which claims that *map* and *filter* commute (which is incorrect as formulated).

$$\begin{aligned}&\forall\, a{:}\,\star, \forall p : a \to \mathit{Bool}, f : a \to a, \mathit{xs} : [\,a\,].\\ &\quad \mathit{map}\ f\ (\mathit{filter}\ p\ \mathit{xs}) == \mathit{filter}\ p\ (\mathit{map}\ f\ \mathit{xs})\end{aligned}$$

A typical QuickCheck user may pick a to be $\mathbb{N}$, and running QuickCheck might produce the following counterexample[1]:

[1] Using a recent QuickCheck extension to show functions.

$p = \{1 \to \mathit{True}, \_ \to \mathit{False}\}; f = \{\_ \to 1\}; xs = [3]$

In other words, if p is a predicate that holds only for 1, and f is the constant function 1, and if we start with a list $[3]$, the property does not hold.

Investigating the left- and right-hand sides as functions from p, f, and xs to lists, we see that an element of type a may either directly come from the list xs, or be the result of applying f. Expressing this in terms of a datatype, we get:

$\mathbf{data}\, A = X\, \mathbb{N} \mid F\, A$

And the property turns into:

$$\forall\, p : A \to \mathit{Bool}, n : \mathbb{N}.\ \mathbf{let}\ xs = [X\, 1\,..\,X\, n]$$
$$\mathbf{in}\ \ \mathit{map}\, F\, (\mathit{filter}\, p\, xs) == \mathit{filter}\, p\, (\mathit{map}\, F\, xs)$$

The only arguments we need to quantify over are the predicate p and the length of the list xs: the function f is fixed to the constructor F. But there is one more advantage; the counterexample that is produced is more descriptive:

$p = \{F\, (X\, 1) \to \mathit{True}, \_ \to \mathit{False}\}; f = F; xs = [X\, 1]$

We clearly see that p holds only for the result of applying f to the (only) element in the list xs.

3 Generalisation

In this section we present a systematic formulation of our schema to test polymorphic functions. Additionally we expose the main theoretical results that back up the method and argue for their correctness. We assume familiarity with basic notions of category theory, notably the interpretation of data types as initial algebras [4, ch. 2].

3.1 Revisiting *reverse*

We start by going through all the necessary steps for one particular concrete example, namely testing two implementations of *reverse* against each other:

$\mathit{reverse1}, \mathit{reverse2} : \forall a.\ [a] \to [a]$

The method we use makes a clear distinction between *arguments* (values that are passed to the function) and *results* (values which are delivered by the function, and should be compared with other results). Furthermore, the arguments are divided up into two kinds; arguments that can be used by the function to *construct* elements of type a, and arguments that can only be used to *observe* arguments of type a.

The first step we take in order to compute the monomorphic instance is to transform the function under test into a function that makes these three parts of the function type explicit. The final type we are looking for is of the form:

$\forall\, a.\ (F\, a \to a) \times (G\, a \to X) \to H\, a$

for functors F, G, H and a monomorphic type X. The argument of type $F\, a \to a$ can be used to construct elements of type a, the argument of type $G\, a \to X$ can be used to observe arguments of type a (by transforming them into a known type X), and $H\, a$ is the result of the function. We call the type above the *canonical*

testing type; all polymorphic functions of the above type can be tested using our method, if there exists an initial F-algebra.

How do we transform functions like *reverse* into functions with a canonical testing type? We start by "dissecting" arguments that can produce as into functions that produce exactly one a. For *reverse*, the one argument that contains as is of type $[a]$. We now make use of the fact that all lists can be represented by a pair of its length and its indexing function, and we thus replace the list argument with an argument of type $\mathbb{N} \times (\mathbb{N} \to a)$ (we will say more about this transformation in section 3.5). After re-ordering the arguments the new type is:

$\forall\, a.\ (\mathbb{N} \to a) \times \mathbb{N} \to [a]$

which fits the requirement, with $F\ a = \mathbb{N}$, $G\ a = ()$, $X = \mathbb{N}$, and $H\ a = [a]$.

For the original function *reverse1* (and similarly for *reverse2*), we can define a corresponding function with a canonical testing type as follows:

$reverse1' : \forall a.\ (\mathbb{N} \to a) \times \mathbb{N} \to [a]$

$reverse1' = reverse1 \circ project$

This uses an auxiliary function to project the arguments of the new function to the initial arguments:

$project : (\mathbb{N} \to a) \times \mathbb{N} \to [a]$

$project\ (x, obs) = map\ x\ [1\,..\,obs]$

Observe that if the new arguments properly cover the domain $(\mathbb{N} \to a) \times \mathbb{N}$, then the original arguments also properly cover the domain $[a]$. It means that the transformations that we have performed to fit the canonical testing type do not weaken the verification procedure.

What have we gained by this rewriting? Our main result says: to test whether two polymorphic functions with a canonical testing type are equal, it is enough to test for equality on the monomorphic type A, where A is the least fixpoint of the functor F, and to use the initial algebra $\alpha : F\ A \to A$ as the first argument.

For the *reverse* example, the least fixpoint of F is simply $\mathbb{N}$ and the initial algebra is the identity function. Thus, to check if $reverse1'$ and $reverse2'$ are equal, we merely have to check

$\forall\, obs : \mathbb{N}.\ reverse1'\ (id, obs) == reverse2'\ (id, obs)$

Writing the transformation explicitly is cumbersome, and indeed we can avoid it, by picking arguments directly from the image of the partially applied projection function, that is, from the set $\{\, project\ (id, obs) \mid obs \in \mathbb{N} \,\}$. By doing so, we obtain the property given in the introduction.

$\forall\, n : \mathbb{N}.\ reverse1\ [1\,..\,n] == reverse2\ [1\,..\,n]$

3.2 Overview

In general, given a function of type $\forall a.\ \sigma[a] \to H\ a$, the objective is to construct a type A, and identify a set of arguments of type $\sigma[a := A]$ to test it against. To do so, we proceed with the following three steps.

1. Transform the function to test and its type $(\forall a.\ \sigma[a] \to H\ a)$ into a function with its type in the canonical form $(\forall a.\ (F\ a \to a) \times (G\ a \to X) \to H\ a)$, where F, G, H are functors. This must be done through an embedding-projection

pair $((e, p) : \sigma[a] \subseteq (F\ a \to a) \times (G\ a \to X))$. The purpose is to identify all the ways (for the function) to construct values of type a, and express them as an algebra of the functor F. (Sect. 3.5).
2. Calculate the initial algebra $(\mu F, \alpha)$ of the functor F. Parametricity and initiality implies that fixing the algebra to α and a to μF still covers all cases. Note that the type argument has now been removed. (Sect. 3.3)
3. Re-interpret the fixing of the algebra to α in step 2 in the context of the original type, using the projection produced in step 1. The arguments to test the function on are picked in the set $\{\, p\,(\alpha, s) \mid s \in G\,(\mu F) \to X \,\}$. (Sect. 3.4)

After these steps the type argument is gone, and testing can proceed as usual. We detail the procedure and argue for its validity in the following sections.

3.3 The initial view

In this section we expose and justify the crucial step of our approach: the removal of polymorphism itself. We begin with showing that applications of (some) polymorphic functions can be expressed in terms of a monomorphic case.

Suppose that the polymorphic function has the type $(\forall a.\ (F\ a \to a) \times (G\ a \to X) \to H\ a)$, that is, its only way to construct values of type a are given by an algebra of functor F, (X is a constant type where a cannot appear). Then, instead of passing a given algebra to a polymorphic function, one can pass the initial algebra, and use the catamorphism of the algebra (often called *fold* and denoted $(\!|\_|\!)$ in the sequel) to translate the results. If the function can also observe the values of the polymorphic parameter, then the observation functions passed as argument must be composed with the catamorphism.

By passing the initial algebra, the type parameter is fixed to μF. The applications of the catamorphism handle the polymorphism, effectively hiding it from the function under test. The following theorem expresses the idea formally. Our proof relies on parametricity [29] and properties of initial algebras [4, ch. 2]

Theorem 1. *Let F, G, H be functors and let $f : (\forall a : \star.(F\ a \to a) \times (G\ a \to X) \to H\ a)$. If there is an initial F-algebra $(\mu F, \alpha)$, then*

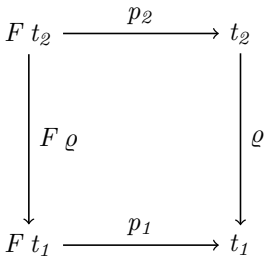
$$\forall\, t : \star, p : F\ t \to t, r : G\ t \to X.\quad f_t\,(p, r) = H\,(\!|p|\!)\,(f_{\mu F}\,(\alpha, r \circ G\,(\!|p|\!)))$$

Proof. We apply the parametricity theorem (restricted to functions) on the type of f, following mechanically the rules given by Fegaras and Sheard [12], theorem 1. After simplification we obtain:

$$\begin{aligned}&\forall\, f : (\forall a : \star.(F\ a \to a) \times (G\ a \to X) \to H\ a),\\ &\quad t_1, t_2 : \star,\quad \varrho : t_2 \to t_1,\\ &\quad p_1 : F\ t_1 \to t_1, p_2 : F\ t_2 \to t_2.\quad r : G\ t_1 \to X,\\ &p_1 \circ F\ \varrho = \varrho \circ p_2 \quad\Rightarrow\quad f_{t_1}\,(p_1, r) = H\ \varrho\,(f_{t_2}\,(p_2, r \circ G\ \varrho))\end{aligned}$$

This equation expresses a general case $(f_{t_1}\,(p_1, r))$ in terms of a specific case $(H\ \varrho\,(f_{t_2}\,(p_2, r \circ G\ \varrho)))$, under the assumption $p_1 \circ F\ \varrho = \varrho \circ p_2$. Here, we hope to find specific values for t_2, q and ϱ which verify the assumption, and obtain a characterisation of the polymorphic case in terms of a monomorphic case.

Satisfying the assumption ($p_1 \circ F\ \varrho = \varrho \circ p_2$) is equivalent to making the diagram on the right commute. Let us pick the following values for t_2, p_2 and ϱ:



- $t_2 = \mu F$, the least fixpoint of F;
- $p_2 = \alpha$, the initial F-algebra;
- $\varrho = (\![p_1]\!)$, the catamorphism of p_1.

We know from properties of initial algebras and catamorphisms that these choices make the diagram commute. Thus, the assumption is verified, and the proof is complete.

Theorem 1 shows that we can express a polymorphic function in terms of a particular monomorphic instance, but the expressions still involve applying (polymorphic) catamorphisms. In the case where we have a function to test (f) and a model (g) to compare against, we can apply theorem 1 to both sides and simplify away the catamorphisms.

Theorem 2. *Let F, G, H be functors, let $f, g{:}\forall a{:}\star.(F\ a \to a) \times (G\ a \to X) \to H\ a$. If there is an initial F-algebra $(\mu F, \alpha)$, then*

$$\begin{array}{lll} & \forall\ s : G\ (\mu F) \to X. & f_{\mu F}\ (\alpha, s) = g_{\mu F}\ (\alpha, s) \\ \Rightarrow & \forall\ a : \star, p : F\ a \to a, r : G\ a \to X. & f_a\ (p, r) = g_a\ (p, r) \end{array}$$

Proof. If $f_{\mu F}\ (\alpha, s) = g_{\mu F}\ (\alpha, s)$ holds for any s, then in particular the equality $f_{\mu F}\ (\alpha, r \circ G\ (\![p]\!)) = g_{\mu F}\ (\alpha, r \circ G\ (\![p]\!))$ holds. Applying $H\ (\![p]\!)$ to both sides of the equality preserves it, and then we can use theorem 1 to transform both sides and obtain that $f_a\ (p, r) = g_a\ (p, r)$ holds for any choice of a, p and r.

3.4 General form of arguments

The results of the previous section apply only to functions of type $(\forall a.\ (F\ a \to a) \times (G\ a \to X) \to H\ a)$. In this section we show that we can extend these results to any argument types which can be *embedded* in $(F\ a \to a) \times (G\ a \to X)$.

Definition 1. *An embedding-projection pair (an EP) is a pair of functions $e : A \to B$, $p : B \to A$ satisfying $p \circ e = id$. Because it constitutes evidence that covering B is enough to cover A, we write $(e, p) : A \subseteq B$ to denote such a pair.*

Given an EP[2] $(e, p) : \sigma[a] \subseteq (F\ a \to a) \times (G\ a \to X)$, one can transform the arguments calculated in the previous section (α paired with any function of type $G\ (\mu F) \to X$) into $\sigma[a]$ by using the projection component, p. The existence of the embedding guarantees that the domain of the original function is properly covered. This idea is expressed formally in the following theorem.

[2] Strictly speaking, this is a polymorphic EP — one EP for each type a.

Theorem 3. *Let F, G, H be functors and let $f, g : \forall a.\ \sigma[a] \to H\ a$. If there is an initial F-algebra $(\mu F, \alpha)$ and an EP $(e, p) : \sigma[a] \subseteq (F\ a \to a) \times (G\ a \to X)$, then*

$$\begin{array}{lll} & \forall\ s : G\,(\mu F) \to X. & f_{\mu F}\,(p\,(\alpha, s)) = g_{\mu F}\,(p\,(\alpha, s)) \\ \Rightarrow & \forall\ a : \star, l : \sigma[a]. & f_a\ l = g_a\ l \end{array}$$

Proof. Apply theorem 2 to $f' = f \circ p$ and $g' = g \circ p$ as follows:

$$\begin{array}{lll} & \forall\ s : G\,(\mu F) \to X. & f_{\mu F}\,(p\,(\alpha, s)) = g_{\mu F}\,(p\,(\alpha, s)) \\ \Leftrightarrow & \quad \{\text{by definition of } \circ,\ f' \text{ and } g'\} & \\ & \forall\ s : G\,(\mu F) \to X. & f'_{\mu F}\,(\alpha, s) = g'_{\mu F}\,(\alpha, s) \\ \Rightarrow & \quad \{\text{by theorem 2}\} & \\ & \forall\ a : \star, q : (F\ a \to a) \times (G\ a \to X). & f'_a\ q = g'_a\ q \\ \Rightarrow & \quad \{\text{by } e\ l \text{ being a special case of } q\} & \\ & \forall\ a : \star, l : \sigma[a]. & f'_a\,(e\ l) = g'_a\,(e\ l) \\ \Leftrightarrow & \quad \{\text{by definition of } \circ,\ f' \text{ and } g'\} & \\ & \forall\ a : \star, l : \sigma[a]. & f_a\,((p \circ e)\ l) = g_a\,((p \circ e)\ l) \\ \Leftrightarrow & \quad \{\text{by the EP law: } p \circ e \equiv id\} & \\ & \forall\ a : \star, l : \sigma[a]. & f_a\ l = g_a\ l \end{array}$$

Properties used for testing are not always expressed in terms of a model, but very often directly as a predicate: they are merely Boolean-valued functions. We can specialise the above result to that case: given a polymorphic predicate, it is enough to verify it for the initial algebra.

Theorem 4. *Let F, G be functors, let $f : \forall a.\ \sigma\,[\,a\,] \to Bool$. If there is an EP $(e, p) : \sigma[a] \subseteq (F\ a \to a) \times (G\ a \to X)$ and an initial F-algebra $(\mu F, \alpha)$, then*

$$\begin{array}{lll} & \forall\ s : G\,(\mu F) \to X. & f_{\mu F}\,(p\,(\alpha, s)) \\ \Rightarrow & \forall\ a : \star, l : \sigma[a]. & f_a\ l \end{array}$$

Proof. Substitute *const True* for g in theorem 3.

One might think that theorem 3, about models, follows from theorem 4, about properties, using $f\,(p, r) = test\,(p, r) == model\,(p, r)$. This is in fact invalid in general, because one cannot assume that equality ($==$) is available for arbitrary types. Indeed, our usage of parametricity in the proof assumes the opposite.

The above results show that it is enough to test on arguments picked from the set $I = \{\, p\,(\alpha, s) \mid s : G\,(\mu F) \to X \,\}$. This could be done by picking elements s in $G(\mu F) \to X$ and testing on $p(\alpha, s)$. However, for the efficiency of testing, it is important *not* to proceed as such, because it can cause redundant tests to be performed. This is because the projection can map different inputs into a single element in I. A better way to proceed is to generate elements of I directly.

3.5 Embedding construction

The previous section shows that our technique can handle arguments that can be embedded in $(F\ a \to a) \times (G\ a \to X)$. In this section we show that all first-order polymorphic arguments can be embedded. Our proof is constructive: it is also a method to build the EP. It is important to construct the embedding because it is used in computing the set of arguments to test the property on.

The general form of a first order argument is a function of type $C\ a \to D\ a$, where C and D are functors and D is polynomial. Note that non-functional values can be represented by adding a dummy argument. Similarly, the above form includes n-ary functions, as long as they are written in an uncurried form. We structure the proof as a series of embedding steps between the most general form and the canonical form. EPs for each step are composed into the final EP. The overall plan is to split all complex arguments into observations or constructors, then group each class together. Lemmas detailing these important steps are given after the top-level proof outline.

Theorem 5. *Let C_i and D_i be functors. If D_i are constructed by sum, products and fixpoints $(0, 1, +, \times, \mu)$, and none of the $C_i\ a$ are empty, then there exist functors F, G and an EP $(e, p) : \forall a : \star.\ \times_i (C_i\ a \to D_i\ a) \subseteq (F\ a \to a) \times (G\ a \to X)$.*

Proof.

$$
\begin{array}{ll}
 & \times_i (C_i\ a \to D_i\ a) \\
\subseteq & \quad \{\text{by lemma 2}\} \\
 & \times_i (C_i\ a \to (S_i \times (P_i \to a))) \\
\equiv & \quad \{\text{by distributing } \to \text{ over } \times\} \\
 & \times_i (C_i\ a \to S_i) \times (C_i\ a \times P_i \to a) \\
\equiv & \quad \{\text{by letting } F_i\ a = G_i\ a \times P_i\} \\
 & \times_i (C_i\ a \to S_i) \times (F_i\ a \to a) \\
\equiv & \quad \{\text{by commutativity and associativity of } \times\} \\
 & \times_i (C_i\ a \to S_i) \times \times_i (F_i\ a \to a) \\
\subseteq & \quad \{\text{by lemma 1}\} \\
 & (G\ a \to X) \times \times_i (F_i\ a \to a) \\
\equiv & \quad \{\text{by } (\tau_1 \to a) \times (\tau_2 \to a) \equiv (\tau_1 + \tau_2) \to a\} \\
 & (G\ a \to X) \times (F\ a \to a)
\end{array}
$$

where $G\ a = \times_i (C_i\ a)$; $X = \times_i X_i$ and $F\ a = +_i (F_i\ a)$.

Lemma 1. *For all types σ_1, σ_2 and non-empty types τ_1, τ_2 ($witness_1 : \tau_1$ and $witness_2 : \tau_2$) then there exists $(e, p) : (\tau_1 \to \sigma_1) \times (\tau_2 \to \sigma_2) \subseteq \tau_1 \times \tau_2 \to \sigma_1 \times \sigma_2$.*

Proof. The embedding applies the embedded functions pair-wise.

$e\ (f_1, f_2) = \lambda(t_1, t_2) \to (f_1\ t_1, f_2\ t_2)$

The projection can be constructed by providing dummy arguments (*witness*) to missing parts of the pair. It is safe to do so, because that part of the pair is ultimately ignored anyway.

$$p\,h = (\lambda t_1 \to fst\ \ (h\,(t_1 \qquad\quad , witness_2)),\\ \qquad\quad \lambda t_2 \to snd\,(h\,(witness_1, t_2 \qquad\quad)))$$

Lemma 2. *Let D be a functor constructed by sum, products and fixpoints. Then there exist types S, P and $(e, p) : D\,a \subseteq S \times (P \to a)$*

Proof. D represents a data structure, which can be decomposed into a shape (S) and a function from positions inside that shape to elements ($P \to a$). (Abbott et al. [1] provide a detailed explanation). The shape can be obtained by using trivial elements ($S = D\,1$). For testing purposes, only structures with a finite number of elements can be generated, and therefore one can use natural numbers for positions ($P = \mathbb{N}$). The projection can traverse the data structure in pre-order and use the second component of the pair ($\mathbb{N} \to a$) to look up the element to put at each position. The corresponding embedding is easy to build.

3.6 Correctness in practice

We have reasoned in a fast-and-loose fashion: our proofs rely on the strongest version of parametricity, which holds only in the polymorphic lambda-calculus.

Applying them to real-world languages (like Ada, Haskell, Java, ML, etc.) is merely "morally correct" [8]. We assume that the functions under test are well-behaved with respect to parametricity: they should not make use of side-effects, infinite data structures, bottoms, etc. In the context of random or exhaustive testing, these assumptions are generally valid. Therefore, our results are readily applicable in practice with a very high level of confidence.

Still, we could extend the result by using a more precise version of parametricity, as for example Johann and Voigtländer [18] expose it.

4 More Examples

4.1 Multiple type parameters

While the theoretical development assumes a single type parameter, we can apply our schema to functions with multiple type parameters. The basic idea is to treat parameters one at a time, assuming the others constant. We do not justify this formally, but merely show how to proceed on a couple of examples.

Example 4 (map). Consider the ubiquitous function *map*, which applies a function to each element in a list.

$map : \forall a\ b.\ (a \to b) \to [a] \to [b]$

As usual, we are interested in testing a candidate *map* function against a known-working model.

We first aim to remove the type parameter a. To do so, we isolate the constructors for a by embedding the list argument into a shape (the length of the list) and a function giving the element at each position (see lemma 2). We obtain the type $\forall a\ b.\ (a \to b) \to \mathbb{N} \to (\mathbb{N} \to a) \to [b]$. We see from the type that the only constructor is an algebra of the functor $F\,a = \mathbb{N}$. The initial F-algebra is

$\mathbf{data}\, A = X\, \mathbb{N}$

After substitution, we have the type $\forall b.\ (A \to b) \to \mathbb{N} \to (\mathbb{N} \to A) \to [b]$, and we know that the third argument is fixed to X.

We can proceed and remove the type parameter b. There is only one constructor for b, which is already isolated, so the initial algebra is easy to compute:

$\mathbf{data}\, B = F\, A$

After substitution, we have the type $(A \to B) \to \mathbb{N} \to (\mathbb{N} \to A) \to [B]$, and we know that the first argument is fixed to F. The second and third arguments can be projected back into a list, so we get the final property:

$$\forall\, n : \mathbb{N}.\ \mathbf{let}\ xs = [\, X\, 1\, ..\, X\, n\,]$$
$$\mathbf{in}\ \ \mathit{map1}\ F\ xs == \mathit{map2}\ F\ xs$$

Note that the function to pass to *map* is fixed: again, only testing for various lengths is enough!

Example 5 (prefix). In Haskell, the standard function *isPrefixOf* tests whether its first argument is a prefix list of its second argument. *isPrefixOf* normally uses the overloaded equality $((==) : a \to a \to \mathit{Bool})$ to compare elements in the first list to elements in the second one. Instead we consider a more general version that explicitly takes a comparison function as parameter. In that case, the types of elements in input lists do not have to match. This generalisation is captured in a type as follows:

$\mathit{isPrefixOf} : \forall a\ b.\ (a \to b \to \mathit{Bool}) \to [a] \to [b] \to \mathit{Bool}$

In this example, the type arguments are completely independent, so we can remove both at once. Both lists can be embedded into a shape ($\mathbb{N}$) and a function from positions ($\mathbb{N} \to a$) in the familiar way. We get the type: $\forall a\ b.\ (a \to b \to \mathit{Bool}) \to \mathbb{N} \to (\mathbb{N} \to a) \to \mathbb{N} \to (\mathbb{N} \to b) \to \mathit{Bool}$.

Computing the initial algebras offers no surprise. We obtain:

$\mathbf{data}\, A = X\, \mathbb{N}$

$\mathbf{data}\, B = Y\, \mathbb{N}$

We have to test functions of type $(A \to B \to \mathit{Bool}) \to \mathbb{N} \to (\mathbb{N} \to A) \to \mathbb{N} \to (\mathbb{N} \to B) \to \mathit{Bool}$, with the third argument fixed to X and the fifth fixed to Y. Again, by using the projection, we know that we can instead generate lists of $X\ i$ and $Y\ j$ to pass directly to the polymorphic function.

Thus, a property to check that two implementations of *isPrefixOf* have the same behaviour is written as follows:

$$\forall\, \mathit{eq} : A \to B \to \mathit{Bool}, m : \mathbb{N}, n : \mathbb{N}.$$
$$\mathbf{let}\ xs = [\, X\, 1\, ..\, X\, m\,]$$
$$\quad ys = [\, Y\, 1\, ..\, Y\, n\,]$$
$$\mathbf{in}\ \ \mathit{isPrefixOf1}\ \mathit{eq}\ xs\ ys == \mathit{isPrefixOf2}\ \mathit{eq}\ xs\ ys$$

What if we had used the less general type $\forall a.\ (a \to a \to \mathit{Bool}) \to [a] \to [a] \to \mathit{Bool}$ (which is isomorphic to the standard type $\forall a.\ \mathit{Eq}\ a \Rightarrow [a] \to [a] \to \mathit{Bool}$)? In that case, the initial algebra would be

$\mathbf{data}\, A = X\, \mathbb{N} \mid Y\, \mathbb{N}$

and the property would look exactly the same. The difference is that the function *eq* would be quantified over a larger set. It would only be passed values

of the form $X\ i$ for the first argument, and $Y\ i$ for the second argument, but the generator of random values does not "know" it. Therefore, it might generate redundant test cases, where *eq* only differs in its results for argument-pairs that are not in the form $X\ i,\ Y\ i$. As we have seen in the above example, this redundancy is avoided by using the most general type. This is another example where more polymorphism makes testing more efficient.

4.2 Assumptions on arguments

It can be quite challenging to write properties for functions whose arguments must satisfy non-trivial properties. For example, generating associative functions or total orders is not obvious. A naive solution is to generate unrestricted arguments and then condition the final property on the arguments being well behaved. This can be highly inefficient if the probability to generate a well-behaved argument is small. Since our technique fixes some parameters, it is sometimes easier to find (or more efficient to generate) arguments with a complex structure. We give examples in the following sections.

Example 6 (Parallel Prefix). A parallel-prefix computation computes the list $[x_1, x_1 \oplus x_2, \ldots, x_1 \oplus \ldots \oplus x_n]$, given an associative operation $\oplus$ and a list of inputs $x_1, \ldots, x_n$. How can we test that two given parallel-prefix computations have equivalent outputs?

We start with the type $\forall a.(a \to a \to a) \to [a] \to [a]$. To isolate the constructors, we rewrite the list type as usual and get $\forall a.(a \to a \to a) \to \mathbb{N} \to (\mathbb{N} \to a) \to [a]$. We can group the constructors to make the algebra more apparent: $\forall a.((a \times a + \mathbb{N}) \to a) \to \mathbb{N} \to [a]$. The next step is to pick the initial algebra.

One might be tempted to use the following datatype and its constructors for the initial algebra.

$\mathbf{data}\ A = \mathit{OPlus}\ A\ A \mid X\ \mathbb{N}$

However, we must take into account that the operation passed to the prefix computation must be associative. The *OPlus* constructor retains too much information: one can recover how the applications of $\oplus$ were associated by examining the structure of A. In order to reflect associativity, a "flat" structure is required. Thus, one should work with lists, as follows:

$\mathbf{type}\ A = [\mathbb{N}]$
$x\ n = [n]$
$\mathit{oplus} = (+\!\!+)$

The final property is therefore:

$\forall\ n : \mathbb{N}.\ \mathbf{let}\ \mathit{xs} = \mathit{map}\ x\ [1\,..\,n]$
$\quad\quad \mathbf{in}\ \mathit{prefix1}\ \mathit{oplus}\ \mathit{xs} == \mathit{prefix2}\ \mathit{oplus}\ \mathit{xs}$

The problem of testing parallel prefix networks has been studied before, notably by Sheeran, who has presented a preliminary version of our result in an invited talk in Hardware Design and Functional Languages [25]. Voigtländer [28] presents another monomorphic instance: he shows that it is enough to test over a 3-value type (**3**). At first sight, it might seem that testing over **3** is better than

over $\mathbb{N}$. However, merely substituting the type-variable with **3** still requires testing all combinations of the other arguments, yielding 113×3^n tests[3] to cover the lists of length n, while by our method a single test is enough for a given length. Again, the efficiency of our method comes from the fixing of more arguments than the type variable.

The above explanation to deal with associativity relies very much on intuition, but it can be generalised. One must always take in account the laws restricting the input when computing the initial algebra: that is, one must find the initial object of the category of algebras that respect those laws. We direct the interested reader to Fokkinga [13] for details.

Example 7 (Insertion in sorted list). Consider testing an insertion function which assumes that its input list is strictly ascending. That is, its type is $\forall a.\ (a \to a \to \mathit{Bool}) \to a \to [a] \to [a]$, but the list argument is restricted to lists that are strictly ascending according to the first argument, which in turn must be a strict total order. After breaking down the list as usual one must handle the type $\forall a.\ (a \to a \to \mathit{Bool}) \to a \to \mathbb{N} \to (\mathbb{N} \to a) \to [a]$.

Forcing the list to be sorted can be tricky to encode as a property of an algebra. So, instead of constraining the lists, we put all the burden on the first argument (an observation): it must be a strict total order that also makes the list ascending. This change of perspective lets us calculate the initial algebra without limitation. We obtain

$\mathbf{data}\ A = Y \mid X\ \mathbb{N}$

The element to insert is Y, and as in many preceding examples, the function receives lists of the form $[X\,1\,..\,X\,n]$. This makes generating suitable orders $(A \to A \to \mathit{Bool})$ easy. Indeed, for such an order (ord) to respect the order of the list, it must satisfy the equation:

$\mathit{ord}\ (X\ i)\ (X\ j) = i < j$

Therefore, we only need to decide on how to order Y with respect to $X\ i$. That is, decide where to position Y in the list. For an input list of length n, there are exactly $n + 1$ possible positions to insert an element. The final property shows how to define the order given a position k for Y.

$$\begin{array}{l}\forall\, n : \mathbb{N}, k : \{0\,..\,n\}.\ \mathbf{let}\ \mathit{xs} = [X\,1\,..\,X\,n] \\ \qquad\qquad\qquad\qquad\ \mathbf{in}\ \ \mathit{insert1}\ (\mathit{ord}\ k)\ Y\ \mathit{xs} == \mathit{insert2}\ (\mathit{ord}\ k)\ Y\ \mathit{xs} \\ \mathbf{where}\ \mathit{ord}\ k\ (X\ i)\ (X\ j) = i < j \\ \qquad\quad \mathit{ord}\ k\ Y\ \ \ Y\ \ \ = \mathit{False} \\ \qquad\quad \mathit{ord}\ k\ (X\ i)\ Y\ \ \ = i \leqslant k \\ \qquad\quad \mathit{ord}\ k\ Y\ \ \ (X\ j) = k < j\end{array}$$

Example 8 (Sorting network). A generator of sorting networks can be represented as a polymorphic function of type $\forall a.\ (a \times a \to a \times a) \to [a] \to [a]$. The first argument is a two-element comparator. Note that, by parametricity,

[3] Voigtländer [28] shows that only some combinations are relevant, but the number of tests is still quadratic in the length of the input list. 113 is the number of associative functions in $\mathbf{3} \to \mathbf{3} \to \mathbf{3}$.

the function cannot check whether the comparator swaps its inputs or not. It is restricted to merely compose instances of the comparator.

Let us apply our schema on the above type. We use the isomorphism $\tau \to a \times b \cong (\tau \to a) \times (\tau \to b)$ to split the first argument, and handle the list as usual. We obtain the following type.

$\forall\, a.\, (a \times a \to a) \to (a \times a \to a) \to \mathbb{N} \to (\mathbb{N} \to a) \to [\,a\,]$

If we overlook the restrictions on the constructors, the initial algebra is

data $A = \mathit{Min}\ A\ A \mid \mathit{Max}\ A\ A \mid X\ \mathit{Int}$

As usual, the sorting function is to be run on $[\,X\,1\,..\,X\,n\,]$. The comparator is built out of *Min* and *Max*. Therefore, to fully test the sorting function, it suffices to test the following function.

$\mathit{sort\_Lat} : \mathbb{N} \to [\,A\,]$

$\mathit{sort\_Lat}\ n = \mathit{sort}\ (\lambda(x, y) \to (\mathit{Min}\ x\ y, \mathit{Max}\ x\ y))\ [\,X\,1\,..\,X\,n\,]$

The output is a list where each element is a comparison tree: a description of how to compute the element by taking minimums and maximums of some elements of the input. In order to verify that the function works, we are left with checking that the output trees are those of a correct sorting function.

Note that this must be checked modulo the laws which restrict our initial algebra. *Min* and *Max* must faithfully represent 2-element comparators which can be passed to the polymorphic function. Therefore, the type A must be understood as a free distributive lattice [10] where *Min* and *Max* are meet ($\wedge$) and join ($\vee$) and Xi are generators.

The correctness of the function can then be expressed as checking each element of the output (o_k) against the output of a known sorting function. Formally:

$$o_k = \bigvee_{M \subseteq \{1 \ldots n\}, \#M = n-k} \left(\bigwedge_{i \in M} Xi \right)$$

There are (at least) two possible approaches to proceed with the verification.

1. Verify the equivalence symbolically, using the laws of the distributive lattice. This is known as the word problem for distributive lattices. One way to do this is to test for syntactical equivalence after transformation to normal form.
2. Check the equivalence for all possible assignments of booleans to the variables $X\ i$, meet and join being interpreted as Boolean conjunction and disjunction. This is valid because truth tables are a complete interpretation of free distributive lattices. In effect, proceeding as such is equivalent to testing the sorting function on all lists of booleans.
   This second way to test equivalence shows that our technique is essentially (at least) as efficient as that of Knuth [19], provided that properties of the distributive lattice structure are cleverly exploited.

5 Related work

Universe-bound polymorphism Jansson et al. [17] have studied the testing of datatype-generic functions: polymorphic functions where the type parameter is

bound to a given universe. This restriction allows them to proceed by case analysis on the shape of the type. In contrast, our method makes the assumption that type parameters are *universally* quantified, taking advantage of parametricity. Since universal quantification and shape analysis are mutually exclusive, Jansson's method and ours complement each other very well.

Shortcut Fusion Shortcut deforestation [14] is a technique to remove intermediate lists. A pre-requisite to shortcut deforestation is that producers of lists are written on the form $g\,(:)\,[\,]$, or essentially, $g\,\alpha$ where α is the initial algebra of the list functor. In general, functions that are normally written in terms of the initial algebra must be parametrised over any algebra, thereby adding a level of polymorphism. This is the exact opposite of the transformation we perform.

Similarity with our work does not stop here, as the correctness argument for shortcut deforestation also relies heavily on polymorphism and parametricity.

Church Encodings The purpose of Church encodings is to encode data types in the pure lambda calculus. Church encodings can also target the polymorphic lambda calculus [5], and the resulting types are polymorphic. In essence, the Church encoding of a data type is the type of its fold (catamorphism). Hinze [15] provides an illuminating example.

Theorem 1 describes (almost) an inverse of Church-encoding: we aim at recovering the datatype underlying polymorphic types. It is not exactly an inverse though: the church-encoded type might be encapsulated in a polymorphic function, which may expose only some of its constructors. Therefore we target these constructors instead of directly targeting the datatype.

Defunctionalisation Reynolds [22] describes defunctionalisation: a transformation technique to remove higher-order functions. Each lambda-abstraction is replaced by a distinctive constructor, whose argument holds the free variables. Applications are implemented via case-analysis: the tag of the constructor tells which which abstraction is entered.

Danvy and Nielsen [9] have shown that defunctionalisation works as an inverse to church encoding. Thus, theorem 1 can be seen as a special case of defunctionalisation, targeted at the constructors of a polymorphic type. However, our main focus is not the removal of function parameters, but of type parameters. Indeed, our embedding step, which *introduces* function parameters, is often crucial for the removal of polymorphism. Note also that we do not transform the function under test. In fact, only the arguments passed to the function are defunctionalised. The constructing functions are transformed to constructors of a datatype, and the observations have to perform case-analysis on this datatype.

Concretisation Pottier and Gauthier [21] introduce *concretisation*: a generalisation of defunctionalisation that can target any source language construct by translating its introduction form into an injection, and its elimination form into case analysis. They apply concretisation to Rémy-style polymorphic records and Haskell type classes, but not removal of polymorphism altogether.

QuickCheck As explained in the introduction, the standard way to test polymorphic functions in QuickCheck [7] is to substitute $\mathbb{N}$ for polymorphic parameters. In the first runs, QuickCheck assigns only small values to parameters of this type, effectively testing small subsets of $\mathbb{N}$. As testing progresses, the size is increased. This strategy is already very difficult to beat! Indeed, we observe that, thanks to parametricity, if one verifies correctness for a type of size n, the function works for all types of size n or less. Additionally, because of the inherent nature of testing, it is only possible to run a finite number of test cases. Therefore, the standard QuickCheck strategy of type instantiation is already very good. We can do better because, in addition to fixing the type, we also fix some (components of) parameters passed to the function. In effect, meaningless tests (tests that are isomorphic to other already run tests, or tests that are unnecessarily specific) are avoided.

criterion	traditional	new
type	$\mathbb{N}$	μF
constructors	$F\mathbb{N} \to \mathbb{N}$	$\{\alpha\}$
observations	$G\mathbb{N} \to X$	$G(\mu F) \to X$

Table 1. Comparison of the traditional QuickCheck praxis to the new method.

The situation is summarised in table 1. By fixing the constructors, a whole dimension is removed from the testing space. Even though the space of observations is enlarged when $\mu F > \mathbb{N}$ (from $G\,\mathbb{N} \to X$ to $G\,(\mu F) \to X$), the trade-off is still beneficial in most cases. We argue informally as follows: if $\mu F > \mathbb{N}$, then F is a "big" functor, such as $F\ a = 1 + a \times a$. This means that the set $F\,\mathbb{N} \to \mathbb{N}$ is big, and as we replace that by a singleton set, this gain dwarfs the ratio between $G\,(\mu F) \to X$ and $G\,\mathbb{N} \to X$, for any polynomial functor G.

Besides efficiency, another benefit to the new method is that the generated counter examples are more informative, as seen on an example in section 2.

In Haskell, there is another pitfall to substituting the polymorphic parameter by $\mathbb{N}$: type classes. Imagine for example that the type parameter is constrained to be an instance of the *Eq* typeclass. Because $\mathbb{N}$ is such an instance, it *is* possible to use it for the type parameter, but this badly skews the distribution of inputs. Indeed, on average, the probability that $a == b$, for generated a and b tends to be very small. A better strategy would be to have a different instance of *Eq* for each run, each with a probability of equality close to $1/2$. Our method does not suffer from this problem: we insist that the methods of classes are explicitly taken into account when identifying the constructors and the observations.

Exhaustive Checking We argue in the previous section that using $\mathbb{N}$ for type parameters is a sensible approach for random testing. However, as Runciman et al. [23] remark, this does not work as well for depth-bound exhaustive testing: the dimension of the test space for constructors ($F\mathbb{N} \to \mathbb{N}$) grows exponentially

as the depth of the search increases. They suggest to use smaller types to test on (such as the unit or Boolean), but the user of the library is left to guess which size is suitable. Our method kills two birds with one stone: we conjure up a suitable type parameter to use, and prevent the exponential explosion of the search for constructors by fixing them. Therefore, we believe that our method is an essential improvement for exhaustive testing of polymorphic functions.

Symbolic execution Tillmann and Schulte [27] generate test cases by symbolic execution of the property to check. As we have mentioned in section 2, our technique can be understood as symbolic execution, therefore, generating test cases by symbolic execution potentially subsumes our method. The advantage of our approach is that it is purely type-based: the monomorphic instance is independent of the actual definition of the property. Therefore, it can work with an underlying black-box tester for monomorphic code.

6 Future work

While the scope of this paper is the testing of polymorphic functions, our technique to remove polymorphism is not specific to testing: any kind of verification technique can be applied on the produced monomorphic instance. This suggests that it has applications outside the domain of testing, maybe in automated theorem proving. This remains to be investigated.

Automated test-case generation libraries typically address the problem of generating random values for monomorphic arguments. We have addressed the problem of calculating values for type arguments. A natural development would be to unify both approaches in the framework of a dependently-typed programming language. A first step towards this goal would be to give a detailed account of parametricity in presence of dependent types.

With the exception of computing initial algebras with laws, the technique described here is completely algorithmic. Therefore, one can assume that it is easy to automate it and build a QuickCheck-like library to test polymorphic properties. However, such a tool would need to analyse the type structure of the functions it is given, and languages based on the polymorphic lambda calculus typically lack such a feature. Moreover, this very feature would invalidate the parametricity theorem, since it relies on universally quantified types being opaque, thereby invalidating our "monomorphisation" transformation. A long term area of research would be to design a programming language where parametricity and type-analysis can be specified on a case-by-case basis. As a short-term goal, we propose to mechanise the technique as an external tool rather than a library, or require the programmer to explicitly inform the polymorphic test generator about the type structure.

We have shown how to get rid of polymorphism using the "initial view" of the type parameters. As there exists a dual to shortcut fusion [26], we conjecture that there exists a dual to our method, using the "final view". That is, the function should be transformed to isolate a co-algebra and fix it to the final element of

the category. Is is unclear at this point what would be the outcome of this dual in terms of testing behaviour.

The technique that we present requires a specific form for the type of the function to test. While our examples show that this form covers a wide range of polymorphic functions that are commonly tested, one can still aspire for a larger applicability. We hope to improve this aspect, either by showing that more types can be embedded, or by amending the core theory. In particular, we address only rank-1 polymorphism: extending to rank-n would be useful. Also, the restriction that F must be a functor in $(F\ a \to a) \times (G\ a \to X)$ seems too specific. Indeed, Church-encoding some types may lead to F being a type-function that is not a functor, and there is *a-priori* no reason that the encoding cannot be reverted. An example is given by Washburn and Weirich [30]: **data** $T = Lam\,(T \to T) \mid App\ T\ T$ is encoded as $\forall a.\ ((a \to a) \to a) \to (a \to a \to a) \to a$, and $F\ a = (a \to a) + (a \times a)$, which is not a functor. We hope to achieve this by fully explaining our technique in a defunctionalisation setting.

7 Conclusion

We have presented a schema for efficient testing of polymorphic properties. The idea is to substitute polymorphic values by a faithful symbolic representation. This symbolic representation is obtained by type analysis, in two steps:

1. isolation of the constructors (yielding a functor F); and
2. restriction to the initial F-algebra.

We suspect that neither of these steps is original, but we could not find them spelt out as such, and therefore we believe that bringing them to the attention of the programming languages community is worthwhile. Furthermore, the testing of polymorphic properties is a novel application for these theoretical ideas.

We have shown on numerous examples, and informally argued that applying our technique not only *enables* testing polymorphic properties by removing polymorphism, but yields good efficiency compared to the standard praxis of substituting $\mathbb{N}$ for the polymorphic argument. In some cases, this improvement is so dramatic that it makes the difference between testing being useful or not. As another evidence of the value of the method, we have applied it to classical problems and have recovered or improved on the corresponding specific results.

Giving a more polymorphic type to a given function enlarges its domain, so one might think that this can increase the amount of testing necessary to verify properties about that function. If our technique is applied, the opposite is true.

You love polymorphism, but you were afraid that it would complicate testing? Fear no more! On the contrary, polymorphism can facilitate testing if approached from the right angle.

Acknowledgments. We would like to give special thanks to Marcin Zalewski, whose repeated interest for the topic and early results gave us the motivation

to pursue the ideas presented in this paper. Josef Svenningson and Ulf Norell provided counter examples to our technique. Peter Dybjer gave useful references about Church encodings. Anonymous reviewers gave useful comments and helped improve the presentation of the paper. This work is partially funded by the Swedish Research Council.

Bibliography

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures*, volume 2620 of *LNCS*, pages 23–38. Springer, Heidelberg, 2003.

[2] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq QuickCheck. In *Proc. of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10. ACM, 2006.

[3] A. H. Bagge, V. David, and M. Haveraaen. Axiom-based testing for C++. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 721–722. ACM, 2008.

[4] R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., 1997.

[5] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39(2-3):135–154, 1985.

[6] J. Christiansen and S. Fischer. EasyCheck — test data for free. In *Functional and Logic Programming*, volume 4989 of *LNCS*, pages 322–336. Springer, 2008.

[7] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM, 2000.

[8] N. A. Danielsson, J. Gibbons, J. Hughes, and P. Jansson. Fast and loose reasoning is morally correct. In *POPL'06*, pages 206–217. ACM Press, 2006.

[9] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proc. of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174. ACM, 2001.

[10] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[11] N. A. Day, J. Launchbury, and J. Lewis. Logical abstractions in Haskell. *In Proc. of the 1999 Haskell Workshop*, 1999.

[12] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 284–294. ACM, 1996.

[13] M. M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6(01):1–32, 1996.

[14] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proc. of the conference on Functional programming languages and computer architecture*, pages 223–232. ACM, 1993.

[15] R. Hinze. Church numerals, twice! *J. of Funct. Program*, 15(01):1–13, 2005.
[16] D. Hoffman, J. Nair, and P. Strooper. Testing generic Ada packages with APE. *Ada Lett.*, XVIII(6):255–262, 1998.
[17] P. Jansson, J. Jeuring, and students of the Utrecht University Generic Programming class. Testing properties of generic functions. In Z. Horvath, editor, *Proceedings of IFL 2006*, volume 4449 of *LNCS*, pages 217–234. Springer-Verlag, 2007.
[18] P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Proc. of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 99–110. ACM, 2004.
[19] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, 2 edition, 1998.
[20] R. Nilsson. ScalaCheck. http://code.google.com/p/scalacheck/, 2009.
[21] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order Symbol. Comput.*, 19(1):125–162, 2006.
[22] J. C. Reynolds. Definitional interpreters for Higher-Order programming languages. *Higher Order Symbol. Comput.*, 11(4):363–397, 1998.
[23] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of the first ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2008.
[24] D. Saff. Theory-infected: or how i learned to stop worrying and love universal quantification. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 846–847. ACM, 2007.
[25] M. Sheeran. Hardware design and functional programming: a perfect match. Talk at Hardware Design and Functional Languages, 2007.
[26] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proc. of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37, pages 124–132. ACM, 2002.
[27] N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.
[28] J. Voigtländer. Much ado about two (pearl): a pearl on parallel prefix computation. *SIGPLAN Not.*, 43(1):29–35, 2008.
[29] P. Wadler. Theorems for free! In *Proc. of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.
[30] G. Washburn and S. Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *Proc. of the eighth ACM SIGPLAN international conference on Functional programming*, pages 249–262. ACM, 2003.