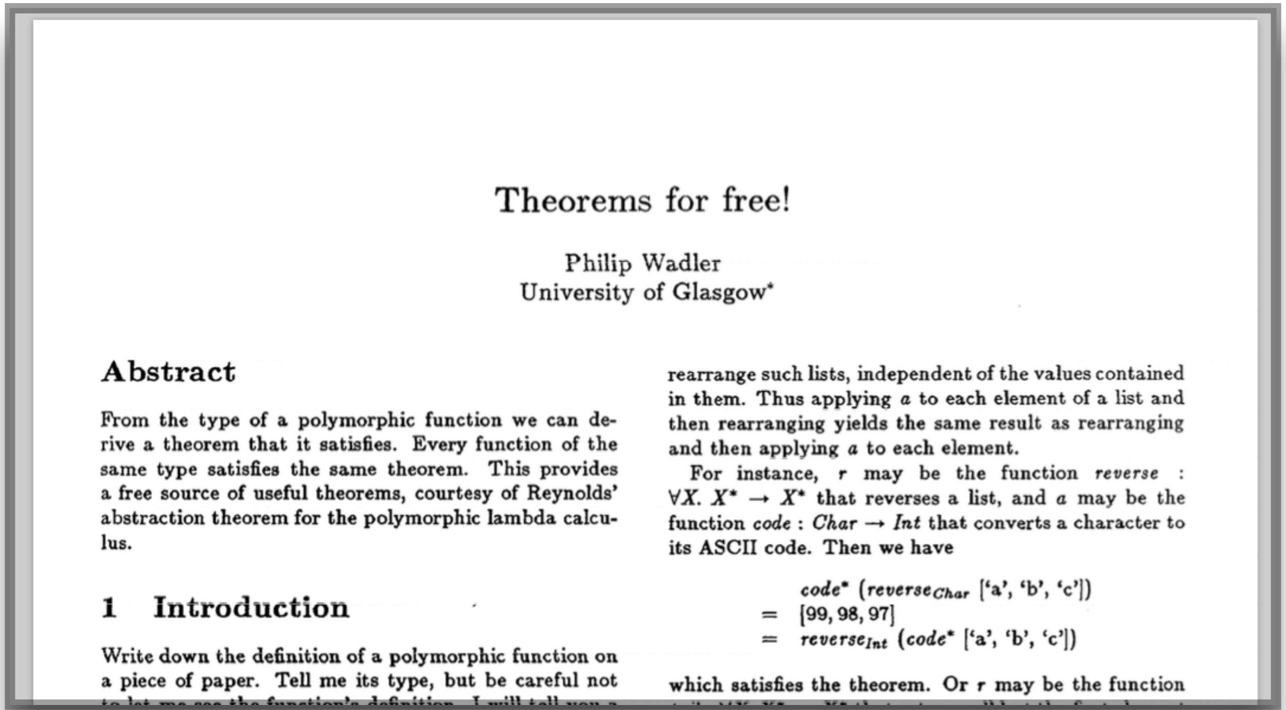
There is no such thing as a free ~~lunch~~ theorem

please excuse the inflammatory title

Parametricity



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

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 $reverse : \forall X.\ X^* \rightarrow X^*$ that reverses a list, and a may be the function $code : Char \rightarrow Int$ 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

Gives us a number of free theorems about parametrically polymorphic functions in *the second order lambda calculus*

Type Classes

Create functions which are not parametrically polymorphic so parametricity says nothing about such functions

Unbounded Recursion

Reduces the number of theorems we can get for free *in a known way*.

Theorems for Free!.
Philip Wadler.
FPCA, New York, New York, USA 1989 pp. 347-359.

Haskell's Seq

Reduces the number of theorems we can get for free *in a known way*.

Theorems for Free!.
Philip Wadler.
FPCA, New York, New York, USA 1989 pp. 347-359.

Type-Indexed Functions

Functions which can make a decision based on the type of an argument.

Kill parametricity dead for the whole language

Everyone you talk to about Type Indexed Functions

So what's with the title?

Type indexed functions are *valuable*. Thus keeping parametricity *costs us* our valuable feature. In fact, parametricity only works in systems where the type system is *very conservative with parametric polymorphism.*

A safe but non-parametric function

Be generous with syntax please, squint your eyes and think, “what could this mean in a Haskell-ish language?”

```
incOrElse n :: Int = n + 1

incOrElse c :: Char = char ((ord c) + 1)

incOrElse o = o
```

```
incOrElse n :: Int = n + 1

incOrElse c :: Char = char ((ord c) + 1)

incOrElse o = o
```

Nothing can go wrong with `incOrElse` if we give it the type $\forall\alpha.\alpha \rightarrow \alpha$

Except, that is, for the theorems we got from parametricity.

So a conservative type system is a *necessary cost* of parametricity

That’s it?

So I have described a problem, you hope I have more than this right?

The solution

Give `incOrElse` some other type - not $\forall \alpha . \alpha \to \alpha$

As long as the type I give it is *compatible* with $\forall \alpha . \alpha \to \alpha$

Suggestion

The types α can be

$$\forall\alpha.\alpha \to \alpha \equiv \forall\alpha \in \boxed{\{\_\}}.\alpha \to \alpha$$

The unknown type

When a function is polymorphic over the unknown type, it is parametrically polymorphic.

Suggestion

```
incOrElse n :: Int = n +1

incOrElse c :: Char = char ((ord c) + 1)

incOrElse o = o
```

$$\forall \alpha \in \boxed{\{\_, \mathrm{Int}, \mathrm{Char}\}}.\alpha \to \alpha$$

When a function is polymorphic over more than the unknown type, it is not parametrically polymorphic.

Future Work

(1) Proof it does not break parametricity in the whole language

(2) Type Inference for such a type system

Why I think (1) is true

The set of types clearly identifies which functions are candidates to have theorems written about them

The parametric arguments are always dispatched in the theorem, e.g. `a . tail = tail . (map a)`

Why I think (2) is true

I have drawn inference outlines and it *seems* to work

It feels like it will work