Abstraction and program design, or the power of parametricity

BOB 2025

Andres Löh

2025-03-14

Well-Typed
The Haskell Consultants

```
(Int, Int) -> (Int, Int)
```

```
(a, a) -> (a, a)
```

```
(a, b) -> (b, a)
```

Well-Typed

Parametric polymorphism, informally

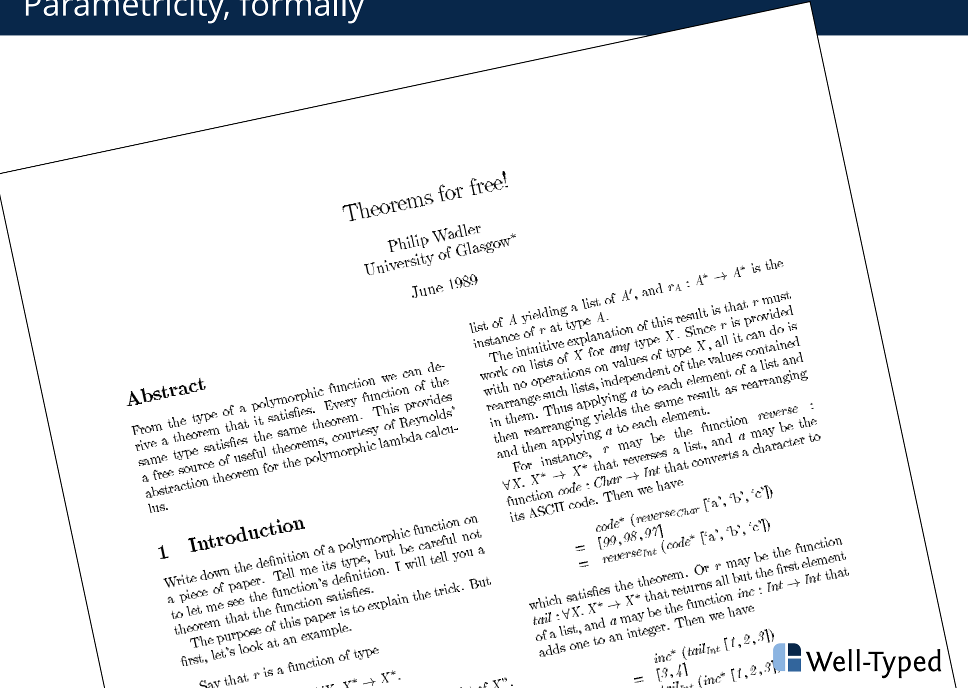
A function is **parametrically polymorphic** in a type variable if it behaves **the same** regardless of instantiation of the type variable.

Parametric polymorphism, informally

A function is **parametrically polymorphic** in a type variable if it behaves **the same** regardless of instantiation of the type variable.

Parametricity then refers to us being able to make (non-trivial) statements about programs by knowing nothing more than their type.

Parametricity, formally



Theorems for free!

Philip Wadler
University of Glasgow*

June 1989

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

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

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

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

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

Parametricity, formally

Not the focus of today's talk.

What **is** the focus?

- Parametric polymorphism grants us practical reasoning capabilities.
- Abstraction can make programs easier to understand.
- Parametric polymorphism can help us design better programs.

Good abstraction / bad abstraction

Bad abstraction?

```haskell
class Transform a where
  transform :: a -> a

instance Transform Int where
  transform :: Int -> Int
  transform i = i + 1

instance Transform String where
  transform :: String -> String
  transform s = map toUpper s
```

Bad abstraction?

```haskell
class Transform a where
  transform :: a -> a

instance Transform Int where
  transform :: Int -> Int
  transform i = i + 1

instance Transform String where
  transform :: String -> String
  transform s = map toUpper s
```

Rule of thumb

Overuse of **ad-hoc** polymorphism makes programs **harder** to understand.

On the other hand …

```
f1 :: a -> a
```

vs.

```
f2 :: Int -> Int
f3 :: String -> String
```

On the other hand …

```
f1 :: a -> a
```

vs.

```
f2 :: Int -> Int
f3 :: String -> String
```

Rule of thumb

Judicious use of **parametric** polymorphism makes programs **easier** to understand.

When can we rely on parametricity?

- No run-time type information.

When can we rely on parametricity?

- No run-time type information.
- As much as possible, restricted side effects.

Well-Typed

When can we rely on parametricity?

- No run-time type information.
- As much as possible, restricted side effects.

- Even in Haskell, we have to consider the presence of crashing and looping computations, and we cannot make statements about performance.

More examples

```
    (b -> c)
 -> (a -> b)
 -> a
 -> c
```

Well-Typed

```
String -> a
```

Well-Typed

```
String -> a

Exception e => e -> IO a
```

```
     (a -> b)
  -> [a]
  -> [b]
```

```
f :: (a -> b) -> [a] -> [b]
```

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function.

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function.
- We can only apply the function if we have a s.

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function.
- We can only apply the function if we have a s.
- We can only obtain a s from the list.

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function.
- We can only apply the function if we have a s.
- We can only obtain a s from the list.

- What can we say about `f []`?

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function.
- We can only apply the function if we have a s.
- We can only obtain a s from the list.

- What can we say about `f []`?
  (Must be `[]`.)
- What can we say about `f (* 2)`?

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function.
- We can only apply the function if we have a s.
- We can only obtain a s from the list.

- What can we say about `f []`?
  (Must be `[]`.)
- What can we say about `f (* 2)`?
  (Must produce a list of even numbers.)
- If `f id [1, 2, 3]` is `[3, 1]`, then what can we say about
  `f g [`x_1`, `x_2`, `x_3`]`?

```
f :: (a -> b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function.
- We can only apply the function if we have a s.
- We can only obtain a s from the list.

- What can we say about `f []`?
  (Must be `[]`.)
- What can we say about `f (* 2)`?
  (Must produce a list of even numbers.)
- If `f id [1, 2, 3]` is `[3, 1]`, then what can we say about
  f g [x_1, x_2, x_3]?
  (Must produce [g x_3, g x_1].)

```
f :: (a -> Bool) -> [a] -> [a]
```

```
f :: (a -> Bool) -> [a] -> [a]
```

- We must produce a s.

Well-Typed

```
f :: (a -> Bool) -> [a] -> [a]
```

- We must produce a s.
- We can only obtain a s from the list.

Well-Typed

```
f :: (a -> Bool) -> [a] -> [a]
```

- We must produce a s.
- We can only obtain a s from the list.
- We can take the result of the function into account . . .

```
f :: (a -> Maybe b) -> [a] -> [b]
```

Well-Typed

```
f :: (a -> Maybe b) -> [a] -> [b]
```

- We must produce b s.

```haskell
f :: (a -> Maybe b) -> [a] -> [b]
```

- We must produce `b` s.
- We can only obtain `b` s by applying the function (and if the test succeeds).

```haskell
f :: (a -> Maybe b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function (and if the test succeeds).
- We can only apply the function if we have a s.

Well-Typed

```haskell
f :: (a -> Maybe b) -> [a] -> [b]
```

- We must produce b s.
- We can only obtain b s by applying the function (and if the test succeeds).
- We can only apply the function if we have a s.
- We can only obtain a s from the list.

Making type signatures more informative

```
f :: (a -> Bool) -> [a] -> [a]
```

vs.

```
f :: (a -> Maybe b) -> [a] -> [b]
```

Well-Typed

Making type signatures more informative

```
f :: (a -> Bool) -> [a] -> [a]
```

vs.

```
f :: (a -> Maybe b) -> [a] -> [b]
```

Similarly:

```
g :: (a -> Bool) -> ([a], [a])
```

vs.

```
g :: (a -> Either b c) -> ([b], [c])
```

```
   IO a
-> (a -> IO b)
-> (a -> IO c)
-> IO c
```

Well-Typed

Conclusions

- Parametrically polymorphic types tell you more than you might think.
- Functions become easier to understand.
- We can try to exploit that when designing our own libraries.
- All this is **not** generally true for abstraction based on ad-hoc polymorphism / type classes, type families, ...