Parametricity

Types Are Documentation

Tony Morris

Danielsson, Hughes, Jansson & Gibbons [DHJG06] tell us:

> *Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.*

Philip Wadler [Wad89] tells us:

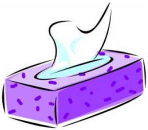
> *Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.*
>
> *The purpose of this paper is to explain the trick.*

Scala

- We will use the Scala programming language for code examples
- However, the point of this talk does not relate to Scala specifically

Scala

Other languages and syntax may be used to denote important concepts and ensure clarity



Why Scala?

- Scala is a legacy hack used primarily by Damo for ciggy-butt brain programming
- Yet it is capable of achieving a high degree of code reasoning
- Speak up if unfamiliarity of syntax inhibits understanding

The Parametricity Trick

This will only work if...

- you write computer programs with inveterate exploitation of the functional programming thesis
- you understand that anything else is **completely insane**
- and if you don't, you're just being a wrong person

NICTA

The Parametricity Trick

This will only work if. . .

- you write computer programs with inveterate exploitation of the functional programming thesis
- you understand that anything else is **completely insane**
- and if you don't, you're just being a wrong person

Reminder

So what is functional programming?

- a means of programming by which expressions are *referentially transparent*.
- but what is referential transparency?

NICTA

Reminder

So what is functional programming?

- a means of programming by which expressions are *referentially transparent*.
- but what is referential transparency?

Referential Transparency

- referential transparency is a potential property of expressions
- functions provide users with referentially transparent expressions

The Test for Referential Transparency

An expression `expr` is referentially transparent if in all programs `p`, all occurrences of `expr` in `p` can be replaced by the result assigned to `expr` without causing an observable effect on `p`.

Referential Transparency

- referential transparency is a potential property of expressions
- functions provide users with referentially transparent expressions

The Test for Referential Transparency

An expression `expr` is referentially transparent if in all programs `p`, all occurrences of `expr` in `p` can be replaced by the result assigned to `expr` without causing an observable effect on `p`.

Referential Transparency

- referential transparency is a potential property of expressions
- functions provide users with referentially transparent expressions

The Test for Referential Transparency

An expression `expr` is referentially transparent if in all programs `p`, all occurrences of `expr` in `p` can be replaced by the result assigned to `expr` without causing an observable effect on `p`.

NICTA

Referential Transparency

Example program

```
p = {
  result = expr
  result = expr
  f(expr, expr)
}
```

Refactoring of program

```
p = {
  f(result, result)
}
```

Is the program refactoring observable for all values of `f`?

Referential Transparency

Example program

```
p = {
  result = expr
  result = expr
  f(expr, expr)
}
```

Refactoring of program

```
p = {
  f(result, result)
}
```

Is the program refactoring observable for all values of `f`?

Referential Transparency

Example program

```
p = {
  result = expr
  result = expr
  f(expr, expr)
}
```

Refactoring of program

```
p = {
  f(result, result)
}
```

Is the program refactoring observable for all values of `f`?

Functional Programming

FP is a commitment to preserving referential transparency

Suppose we encountered the following function definition:

```
def add10(n: Int): Int
```

By the type alone, there are $(2^{32})^{2^{32}}$ possible implementations

Lossful Reasoning

Sacrificing efficiency to gain unreliability

We might form a suspicion that `add10` adds ten to its argument

```
def add10(n: Int): Int
```

So we write some tests:

```
add10(0)        = 10
add10(5)        = 15
add10(-5)       = 5
add10(223)      = 233
add10(5096)     = 5106
add10(2914578)  = 29145588
add10(-2914578) = -29145568
```

And conclude, yes, this function adds ten to its argument

Lossful Reasoning
Sacrificing efficiency to gain unreliability

```
def add10(n: Int): Int =
  if(n < 8000000) n + 10
  else n * 7
```

Wason Rule Discovery Test, *confirmation bias[GB02]*.

We will just write more tests!

```
add10(18916712)  = 18916722
add10(-18916712) = -18916702
```

. . . or we might come up with some system of apologetics for this shortfall

- “A negligent programmer has misnamed this function"
- “More tests will fix it"
- “Well we can’t test everything!"

Lossful Reasoning

Sacrificing efficiency to gain unreliability

We are reinforcing our excess confidence in our belief that we are being responsible programmers

We aren't

Actually, we can do significantly better with a machine-checked proof, mitigating our disposition to biases

Automating "Automated Testing"?

Reasoning with parametricity

Monomorphic Signature

- Examining the signature `Int => Int`
- We see a lot of things this function does *not* do
- For example, it never returns the value `"abc"`
- However, there is an unmanageable number of possible things it might do

Another monomorphic example

- Examining the signature `List[Int] =>List[Int]`
- For example, it might add all the `Ints` and return a list arrangement that depends on whether or not the result is a prime number
- The possibilities are *enormous*

Reasoning with parametricity

Polymorphic Signature

```
def irrelevant[A](x: List[A]): List[A]
```

- We can immediately assert, with confidence, a lot of things about how this function works *because it is polymorphic*
- More directly, we assert what the function does not do
- In other words, *parametricity* has improved readability
- Really? By how much?

```
List<A> irrelevant<A>(List<A> x) // C#
```

```
<A> List<A> irrelevant(List<A> x) // Java
```

NICTA

Reasoning with parametricity

Polymorphic Signature

```
def irrelevant[A](x: List[A]): List[A]
```

- We can immediately assert, with confidence, a lot of things about how this function works *because it is polymorphic*
- More directly, we assert what the function does not do
- In other words, *parametricity* has improved readability
- Really? By how much?

```
List<A> irrelevant<A>(List<A> x) // C#
```

```
<A> List<A> irrelevant(List<A> x) // Java
```

Reasoning with parametricity

```scala
def irrelevant[A](x: List[A]): List[A] =
  ...
```

Theorem

Every element A in the result list appears in the input.
Contraposed, If A is not in the input, it is not in the result

```
List<A> irrelevant<A>(List<A> x) // C#
```

```
<A> List<A> irrelevant(List<A> x) // Java
```

Reasoning with parametricity

I know this because . . .

- Because I am the boss and I said so
- Because Reliable Rob told me so
- Because the *function name* told me so
- Because the comment told me so
- **Because it would not have compiled otherwise**

NICTA

Reasoning with parametricity

I know this because ...

- Because I am the boss and I said so
- Because Reliable Rob told me so
- Because the *function name* told me so
- Because the comment told me so
- **Because it would not have compiled otherwise**

Reasoning with parametricity

I know this because . . .

- Because I am the boss and I said so
- Because Reliable Rob told me so
- Because the *function name* told me so
- Because the comment told me so
- **Because it would not have compiled otherwise**

Reasoning with parametricity

I know this because . . .

- Because I am the boss and I said so
- Because Reliable Rob told me so
- Because the *function name* told me so
- Because the comment told me so
- **Because it would not have compiled otherwise**

NICTA

Reasoning with parametricity

I know this because . . .

- Because I am the boss and I said so
- Because Reliable Rob told me so
- Because the *function name* told me so
- Because the comment told me so
- **Because it would not have compiled otherwise**

Reasoning with parametricity

Uninhabited Example

```
def irrelevant[A, B](a: A): B =
  ...
```

Theorem

*This function **never** returns because if it did, it would never have compiled*

```
List<B> irrelevant<A, B>(List<A> x) // C#
```

```
<A, B> List<B> irrelevant(List<A> x) // Java
```

Reasoning with parametricity

Fast and loose reasoning is morally correct [DHJG06]

Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.

What does this mean exactly?

Fast and Loose Reasoning

```scala
def even(p: Int): Boolean =
  ...
```

Theorem

The `even` *function returns either* `true` *or* `false`

```
bool even(int p) // C#
```

```
boolean even (int p) // Java
```

Fast and Loose Reasoning

```
def even(p: Int): Boolean =
  even(p)
```

Actually, the `even` function doesn't even return, *yet we casually exclude this possibility in discussion*.

Fast and Loose Reasoning

Scala has a ~~few~~ lot of undermining escape hatches

- `null`
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

```
def irrelevant[A](x: List[A]): List[A] =
  null
```

Theorem

Every A element in the result list appears in the input list

Well, not if you don't even return a list. `null` **breaks parametricity.**

```
def irrelevant[A](x: A): Boolean =
  x.isInstanceOf[Int] ||
  x match {
    case (s: String) => s.length < 10
  }
```

Theorem

This function ignores its argument and consistently returns either `true` *or* `false`

Type-casing[1] breaks parametricity

[1] case-analysis on type

```
def irrelevant[A](x: List[A]): List[A] =
  "abc".asInstanceOf[A] :: x
```

Theorem

Every A element in the result list appears in the input list

Type-casting breaks parametricity

Fast and Loose Reasoning
side-effect escape hatch

```
def irrelevant[A](x: A): A = {
    println("hi")
    x
  }
```

Theorem

This function only ever does one thing —return its argument

Side-effects breaks parametricity

```
def irrelevant[A](x: A): Int =
  x.toString.length
```

Theorem

This function ignores its argument to return one of 2^{32} values.

Java's `Object` methods break parametricity

```
def reverse[A, B](x: List[A]): List[B] =
  x.foldLeft[List[B]](Nil)((b, a) =>
    a.asInstanceOf[B] :: b)
```

Theorem

This function ***always*** *returns `Nil` and so cannot possibly reverse the list*

Type-casting breaks parametricity

Fast and Loose Reasoning

- Scala sure does have a lot of escape hatches!
- if we abandon all these escape hatches, to what extent is the programming environment disabled?

NICTA

Fast and Loose Reasoning

- For example, Haskell disables side-effects, type-casing and type-casting, *giving a significant advantage for no penalty*
- so what about Scala?
- can we use a reliable subset without too much penalty?

NICTA

Yes.

And we do.

The Scalazzi Safe Scala Subset

- ~~`null`~~
- ~~exceptions~~
- ~~Type-casing (`isInstanceOf`)~~
- ~~Type-casting (`asInstanceOf`)~~
- ~~Side-effects~~
- ~~`equals`/`toString`/`hashCode`~~
- ~~`notify`/`wait`~~
- ~~`classOf`/`.getClass`~~
- General recursion

The Scalazzi Safe Scala Subset

- We have now **improved** our reasoning abilities, but at what cost?
- It turns out that eliminating these escape hatches results in a **significant language improvement** with minimal, orthogonal, easily-managed penalties
- In other words, we can assume the language subset absent these attributes and by doing so, achieve a large net benefit

Fast and Loose Reasoning

The Scalazzi Safe Scala Subset

- We have now **improved** our reasoning abilities, but at what cost?
- It turns out that eliminating these escape hatches results in a **significant language improvement** with minimal, orthogonal, easily-managed penalties
- In other words, we can assume the language subset absent these attributes and by doing so, achieve a large net benefit

NICTA

The Scalazzi Safe Scala Subset

- We have now **improved** our reasoning abilities, but at what cost?
- It turns out that eliminating these escape hatches results in a **significant language improvement** with minimal, orthogonal, easily-managed penalties
- In other words, we can assume the language subset absent these attributes and by doing so, achieve a large net benefit

- Some open-source projects, using Scala, even Java and C#, apply fast and loose reasoning to achieve confidence in the excellence of other team members
- Project contributors rarely step on each others' (or their own) toes precisely because of this optimistic approach
- Cynics fail hard

- Some open-source projects, using Scala, even Java and C#, apply fast and loose reasoning to achieve confidence in the excellence of other team members
- Project contributors rarely step on each others' (or their own) toes precisely because of this optimistic approach
- Cynics fail hard

- Some open-source projects, using Scala, even Java and C#, apply fast and loose reasoning to achieve confidence in the excellence of other team members
- Project contributors rarely step on each others' (or their own) toes precisely because of this optimistic approach
- Cynics fail hard

Fast and Loose Reasoning
It works

Parametricity is principled and it works

Tell me again about this "real world."

Scaling Parametricity

```
def forallM[F[_]: Monad, A]
  (p: A => F[Boolean], o: Option[A]): F[Boolean]
```

Theorem

The `Boolean` *result depends on zero or more of*

- *None of its arguments*
- *Whether the* `Option` *is a* `Some` *or* `None`
- *If the* `Option` *is a* `Some`*, then the result of having applied the given function to the* `Some` *value*
- *Multiple applications of sequencing of the effect (*`F[Boolean]`*) in the* `Some` *case*
- *in other words, one of (2 * 2 * 2) inhabitants before accounting for multiple effect sequencing*

Scaling Parametricity

We conclude that, discounting multiple effect sequencing, there are 8 possible inhabitants 1:

1. always `false`
2. always `true`
3. `o.isDefined`
4. `o.isEmpty`
5. `Some(a) => p(a) else false`
6. `Some(a) => p(a) else true`
7. `Some(a) => !p(a) else false`
8. `Some(a) => !p(a) else true`

Scaling Parametricity

Importantly

The implementation may only use the monad primitive operations, even though the use-case may apply a specific monad context. If it were a specific monad (e.g. `F=List`), the inhabitants become wildly unmanageable and the value of using the type for documentation hovers ever closer to zero.

Scaling Parametricity

For example

The `forallM` function definitely does not perform any IO effects (`F=IO`), even though the function user may apply that specific use-case

and so on . . .

The Limits of Parametricity

```
def thisIsNotReverse[A](x: List[A]): List[A]
```

OK, so we know that all elements in the result appear in the input

- but how do we narrow it down?
- how do we rule out all possibilities for the type but one?
- how do we specifically determine what the function does?

By types (proof) alone, it is not possible to narrow down to one possibility in the *general case*

However

- We can provide once-inhabitance for some specific cases
- Types are proof-positive
- We have tools to assist us when we come up against these limitations
- Tests are failed proof-negative

Produce an implementation that does **not** reverse

```
module ThisMightReverse where

-- | This function does not reverse.
--
-- >>> thisMightReverse []
-- []
--
-- prop> (thisMightReverse . thisMightReverse) x == x
--
-- prop> thisMightReverse (x ++ y) == (thisMightReverse y ++ thisMightReverse x)
thisMightReverse ::
  [Int]
  -> [Int]
thisMightReverse =
  error "todo"
```

Produce an implementation that does **not** reverse

```haskell
module ThisMightReverse where

-- | This function does not reverse.
--
-- >>> thisMightReverse []
-- []
--
-- prop> (thisMightReverse . thisMightReverse) x == x
--
-- prop> thisMightReverse (x ++ y) == (thisMightReverse y ++ thisMightReverse x)
thisMightReverse ::
  [Int]
  -> [Int]
thisMightReverse =
  let sw i | even i = i + 1
           | otherwise = i - 1
  in foldl (flip (:)) [] . map sw
```

Produce an implementation that does **not** reverse

```
module ThisMightReverse where

-- | This function does not reverse.
--
-- >>> thisMightReverse []
-- []
--
-- prop> (thisMightReverse . thisMightReverse) x == x
--
-- prop> thisMightReverse (x ++ y) == (thisMightReverse y ++ thisMightReverse x)
thisMightReverse ::
  [a]
  -> [a]
thisMightReverse =
  error "todo"
```

We can't!

The function has been fully-specified by:

- The parametric type
- Tests

The function, `thisMightReverse` definitely reverses the list **without looking at the source code or the function name**

Parametricity

Parametricity is . . .

an efficient, reliable tool to assist code-readability to assist creating non-trivial software in a team environment.

Parametricity

Fast and loose reasoning is morally correct

Identifier-name reasoning is morally obnoxious

References

- Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons, *Fast and loose reasoning is morally correct*, ACM SIGPLAN Notices, vol. 41, ACM, 2006, pp. 206–217.
- Maggie Gale and Linden J Ball, *Does positivity bias explain patterns of performance on wason's 2-4-6 task?*
- Philip Wadler, *Theorems for free!*, Proceedings of the fourth international conference on Functional programming languages and computer architecture, ACM, 1989, pp. 347–359.

NICTA

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ _ = return False
```

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ _ = return True
```

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ Nothing = return False
forallM _ (Just _) = return True
```

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ Nothing = return True
forallM _ (Just _) = return False
```

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ Nothing = return False
forallM p (Just a) = p a
```

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ Nothing = return True
forallM p (Just a) = p a
```

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ Nothing = return False
forallM p (Just a) = p a >>= return . not
```

```
forallM :: Monad m => (a -> m Bool) -> Maybe a -> m Bool
forallM _ Nothing = return True
forallM p (Just a) = p a >>= return . not
```