Dysfunctional Programming

András Kovács

Eötvös Loránd University, Dept. of Programming Languages and Compilers

9 February 2023, Department Workshop

First-class functions

A huge amount of software abstraction is based on first-class functions:

- HOFs in FP
- Virtual methods in OOP
- Callbacks, closures in many settings

First-class functions

A huge amount of software abstraction is based on first-class functions:

- HOFs in FP
- Virtual methods in OOP
- Callbacks, closures in many settings

Lots of headache in compilation:

- Dynamic calls block static analysis & inlining
- Implementation of closures/vtables adds complexity

First-class functions

A huge amount of software abstraction is based on first-class functions:

- HOFs in FP
- Virtual methods in OOP
- Callbacks, closures in many settings

Lots of headache in compilation:

- Dynamic calls block static analysis & inlining
- Implementation of closures/vtables adds complexity

What can we do without first-class functions?

Do we need function values here?

```
map : (A → B) → List A → List B
map f nil         = nil
map f (cons a as) = cons (f a) (map f as)

g : List Int → List Int
g = map (+1)
```

Do we need function values here?

```
map : (A → B) → List A → List B
map f nil         = nil
map f (cons a as) = cons (f a) (map f as)

g : List Int → List Int
g = map (+1)
```

Instead:

```
g : List Int → List Int
g nil         = nil
g (cons a as) = cons (a + 1) (g as)
```

Do we need function values here?

```
map : (A → B) → List A → List B
map f nil         = nil
map f (cons a as) = cons (f a) (map f as)

g : List Int → List Int
g = map (+1)
```

Instead:

```
g : List Int → List Int
g nil         = nil
g (cons a as) = cons (a + 1) (g as)
```

Function elimination by inlining.

Getting rid of function values

1. Metaprogramming: use first-class functions only at compile time, generate first-order code. Inlining is a special case of this.
2. Defunctionalization: replace closures by explicit first-order data.

We focus on **1**, using **two-level type theory**.

Getting rid of function values

1. Metaprogramming: use first-class functions only at compile time, generate first-order code. Inlining is a special case of this.
2. Defunctionalization: replace closures by explicit first-order data.

We focus on **1**, using **two-level type theory**.

There is a dependently typed compile-time language for metaprogramming.

Getting rid of function values

1. Metaprogramming: use first-class functions only at compile time, generate first-order code. Inlining is a special case of this.
2. Defunctionalization: replace closures by explicit first-order data.

We focus on **1**, using **two-level type theory**.

There is a dependently typed compile-time language for metaprogramming.

Object-level & meta-level code is freely mixed in programs.

Getting rid of function values

1. Metaprogramming: use first-class functions only at compile time, generate first-order code. Inlining is a special case of this.
2. Defunctionalization: replace closures by explicit first-order data.

We focus on **1**, using **two-level type theory**.

There is a dependently typed compile-time language for metaprogramming.

Object-level & meta-level code is freely mixed in programs.

Generated code is guaranteed to be *well-typed*, *simply typed* and *only have first-order functions*.

Rules, informally

- Compile-time types and runtime types have different types ("universes").
- Function and non-function ("value") types at runtime stage are distinguished.
- There are only first-order function types at runtime.
- We can't store functions in runtime data.

```
data List (A : Ty Val) : Ty Val where
  nil  : List A
  cons : A → List A → List A

map : {A B : Ty Val} → (A → B) → List A → List B

g : List Int -> List Int
g := map (+1)
```

Is List a Functor/Applicative/Monad?

OK:

```
class Functor (F : Ty Val → Ty Val) where
  fmap : (A → B) → F A → F B
```

Is List a Functor/Applicative/Monad?

OK:

```
class Functor (F : Ty Val → Ty Val) where
  fmap : (A → B) → F A → F B
```

Issue with `Applicative`:

```
class Applicative (F : Ty Val → Ty Val) where
  pure : A → F A
  ap   : F (A → B) → F A → F B
```

Is List a Functor/Applicative/Monad?

OK:

```
class Functor (F : Ty Val → Ty Val) where
  fmap : (A → B) → F A → F B
```

Issue with `Applicative`:

```
class Applicative (F : Ty Val → Ty Val) where
  pure : A → F A
  ap   : F (A → B) → F A → F B
```

Solution:

```
class Applicative (F : Ty Val → Ty Val) where
  point : F ()
  zip   : F A → F B → F (A, B)
```

Is List a Functor/Applicative/Monad?

OK:

```
class Functor (F : Ty Val → Ty Val) where
  fmap : (A → B) → F A → F B
```

Issue with `Applicative`:

```
class Applicative (F : Ty Val → Ty Val) where
  pure : A → F A
  ap   : F (A → B) → F A → F B
```

Solution:

```
class Applicative (F : Ty Val → Ty Val) where
  point : F ()
  zip   : F A → F B → F (A, B)
```

Monad is also OK:

```
bind : M A → (A → M B) → M B
```

Is State a Functor/Applicative/Monad?

```
State : Ty Val → Ty Val → MetaTy
State S A = S → (A, S)
```

Using the previous Functor/Applicative/Monad definitions, no!

Is State a Functor/Applicative/Monad?

```
State : Ty Val → Ty Val → MetaTy
State S A = S → (A, S)
```

Using the previous Functor/Applicative/Monad definitions, no!

But it is a *relative* Functor/Applicative/Monad:

```
class RelFunctor (F : Ty Val → MetaTy) where ...
class RelApplicative (F : Ty Val → MetaTy) where ...
class RelMonad (M : Ty Val → MetaTy) where ...
instance RelMonad (State S) where ...
```

Defunctionalization

Difference lists in Haskell:

```haskell
type DList a = [a] -> [a]

append :: DList a -> DList a -> DList a
append xs ys = xs . ys

toDList a as = a : as
fromDList xs = xs []
```

Defunctionalization

No functions needed:

```
data DList a = Chunk [a] | Append (DList a) (DList a)

apply :: DList a -> [a] -> [a]
apply (Chunk xs)     ys = xs ++ ys
apply (Append xs ys) zs = apply xs (apply ys zs)

toDList :: [a] -> DList a
toDList xs = Chunk xs

fromDList :: DList a -> [a]
fromDList xs = apply xs []
```

Summary & further topics

- Not having first-class functions makes compilation much easier.
  - In most cases: metaprogramming can replace first-class functions.
  - In most remaining cases: defunctionalization can replace first-class functions.
  - In the remaining cases: we can distinguish closures *in the type system*.
- Trade-off between programming convenience and performance.
- Trade-off between code size and performance.
- Other things to try to do without functions:
  - Fusion optimizations.
  - More things from the Haskell/ML literature.