High-performance defunctionalization in Futhark

Anders Kiel Hovgaard Troels Henriksen Martin Elsman

Department of Computer Science
University of Copenhagen
(DIKU)

Trends in Functional Programming, 2018

Motivation

- Massively parallel processors, like GPUs, are common but difficult to program.
- Functional programming can make it easier to program GPUs:
  - Referential transparency.
  - Expressing data-parallelism.

Problem Higher-order functions cannot be directly implemented on GPUs.

Can we do higher-order functional GPU programming anyway?

Motivation

Higher-order functions on GPUs?

- Yes!
- Using moderate type restrictions, we can eliminate all higher-order functions at compile-time.
- Gain many benefits of higher-order functions without any run-time performance overhead.

Reynolds's defunctionalization

Defunctionalization (Reynolds, 1972)

John Reynolds: "Definitional interpreters for higher-order programming languages", ACM Annual Conference 1972.

Basic idea:

- Replace each function abstraction by a tagged data value that captures the free variables:

$$\lambda x : \mathbf{int}.\, x + y \quad \Longrightarrow \quad \mathit{LamN}\ y$$

- Replace application by case dispatch over these functions:

$$f\ a \quad \Longrightarrow \quad \begin{array}{l} \mathbf{case}\ f\ \mathbf{of}\ \mathit{Lam1} \ldots \\ \phantom{\mathbf{case}\ f\ \mathbf{of}\ } \mathit{Lam2} \ldots \\ \phantom{\mathbf{case}\ f\ \mathbf{of}\ } \mathit{LamN}\ y \to a + y \\ \phantom{\mathbf{case}\ f\ \mathbf{of}\ } \ldots \end{array}$$

- Branch divergence on GPUs.

Language and type restrictions

Futhark

A purely functional, data-parallel array language with an optimizing compiler that generates GPU code via OpenCL.

- Parallelism expressed through built-in higher-order functions, called *second-order array combinators* (SOACs):

  map, **reduce**, **scan**, ...

- No recursion, but sequential loop constructs:

  ```
  loop pat = init for x in arr do body
  ```

Type-based restrictions on functions

To permit efficient defunctionalization, we introduce type-based restrictions on the use of functions.

Statically determine the form of every applied function.

Transformation is simple and eliminates all higher-order functions.

Instead of allowing unrestricted functions and relying on subsequent analysis, we entirely avoid such analysis.

Type-based restrictions on functions

Conditionals may not produce functions:

```
let f = if b1 then ...
                    if bN then λx → e_n
              else ... λx → e_k
in ... f y
```

Which function `f` is applied?

If our goal is to eliminate higher-order functions without introducing branching, we must restrict conditionals from returning functions.

Require that branches have **order zero** type.

Type-based restrictions on functions

Arrays may not contain functions:

```
let fs = [λy → y+a, λz → z+b, ...]
in ... fs[n] 5
```

Which function `fs[n]` is applied?

Also need to restrict **`map`** to not create array of functions:

```
map (λx → λy → ...) xs
```

Type-based restrictions on functions

Loops may not produce functions:

```
loop f = (λz → z+1) for x in xs
do (λz → x + f z)
```

The shape of `f` depends on the number of iterations of the loop.

Require that loop has **order zero** type.

All other typing rules are standard and do not restrict functions.

Defunctionalization

Defunctionalization

- Type restrictions enable us to track functions precisely.
- Control-flow is restricted so every applied function is known and every application can be specialized.

Defunctionalization

Defunctionalization in a nutshell:

```
let a = 1
let b = 2
let f = λx → x+a
in f b
```

```
let a = 1
let b = 2
let f = {a=a}
in f’ f b
```

Create lifted function:

```
let f’ env x =
  let a = env.a
  in x+a
```

Defunctionalization

Static values:

$$\begin{aligned} sv ::= &\ Dyn\ \tau \\ \mid &\ Lam\ x\ e_0\ E \\ \mid &\ Rcd\ \{(\ell_i \mapsto sv_i)^{i \in 1..n}\} \end{aligned}$$

- Static approximation of the value of an expression.
- Precisely capture the closures produced by an expression.

Translation environment E maps variables to static values.

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let twice = {}
```

$Lam\ g\ (\lambda x \to g\,(g\,x))\ []$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

⇝

```
let twice = {}                              Lam g (λx → g (g x)) []
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let twice = {}
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$Lam\ g\ (\lambda x \to g\,(g\,x))\ [\,]$

$$\begin{array}{rcl} \text{twice} & \rightsquigarrow & \text{twice} \\ (\lambda y \to y + a) & \rightsquigarrow & \{a = a\}, \quad Lam\ y\ (y + a)\ [a \mapsto Dyn\ \mathbf{int}] \end{array}$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let twice = {}                                    Lam g (λx → g (g x)) []
let main = let f = let a = 5
                   in twice' twice {a = a}
           in f 1
let twice' (env: {}) (g: {a: int}) = λx → g (g x)
```

$$\begin{array}{rcl} \text{twice} & \rightsquigarrow & \text{twice} \\ (\lambda y \to y + a) & \rightsquigarrow & \{a = a\}, \quad Lam\ y\ (y + a)\ [a \mapsto Dyn\ \mathbf{int}] \end{array}$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let twice = {}                                    Lam g (λx → g (g x)) []
let main = let f = let a = 5
                   in twice' twice {a = a}
           in f 1
let twice' (env: {}) (g: {a: int}) = λx → g (g x)
```

$$\text{twice} \quad \rightsquigarrow \quad \text{twice}$$

$$(\lambda y \rightarrow y + a) \quad \rightsquigarrow \quad \{a = a\}, \quad \underbrace{\mathit{Lam}\ y\ (y + a)\ [a \mapsto \mathit{Dyn}\ \mathbf{int}]}_{g}$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let twice = {}                                    Lam g (λx → g (g x)) []
let main = let f = let a = 5
                   in twice' twice {a = a}
           in f 1
let twice' (env: {}) (g: {a: int}) = λx → g (g x)
```

$$\begin{array}{rcl}
\text{twice} & \rightsquigarrow & \text{twice} \\
(\lambda y \to y + a) & \rightsquigarrow & \{a = a\},\quad \underbrace{\mathit{Lam}\ y\ (y + a)\ [a \mapsto \mathit{Dyn}\ \mathbf{int}]}_{g} \\
\lambda x \to g\ (g\ x) & \rightsquigarrow & \{g = g\}, \\
 & & \mathit{Lam}\ x\ (g\ (g\ x))\ [g \mapsto \mathit{Lam}\ y\ (y + a)\ \ldots)]
\end{array}$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let twice = {}
let main = let f = let a = 5
                   in twice' twice {a = a}
           in f 1
let twice' (env: {}) (g: {a: int}) = {g = g}
```

$Lam\ g\ (\lambda x \to g\ (g\ x))\ [\,]$

$$
\begin{array}{rcl}
\text{twice} & \rightsquigarrow & \text{twice} \\
(\lambda y \to y + a) & \rightsquigarrow & \{a = a\},\quad \underbrace{Lam\ y\ (y + a)\ [a \mapsto Dyn\ \mathbf{int}]}_{g} \\
\lambda x \to g\ (g\ x) & \rightsquigarrow & \{g = g\}, \\
 & & Lam\ x\ (g\ (g\ x))\ [g \mapsto Lam\ y\ (y + a)\ \ldots)]
\end{array}
$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let twice = {}
let main = let f = let a = 5
                   in {g = {a = a}}
           in f 1
let twice' (env: {}) (g: {a: int}) = {g = g}
```

$Lam\ g\ (\lambda x \to g\ (g\ x))\ []$

$$\begin{aligned}
\text{twice} &\quad\rightsquigarrow\quad \text{twice} \\
(\lambda y \to y + a) &\quad\rightsquigarrow\quad \{a = a\},\quad \underbrace{Lam\ y\ (y + a)\ [a \mapsto Dyn\ \textbf{int}]}_{g} \\
\lambda x \to g\ (g\ x) &\quad\rightsquigarrow\quad \{g = g\}, \\
&\qquad\quad Lam\ x\ (g\ (g\ x))\ [g \mapsto Lam\ y\ (y + a)\ \ldots)]
\end{aligned}$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
⇝

let main = let f = let a = 5
                   in {g = {a = a}}
           in f 1
```

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let main = let f = let a = 5
                   in {g = {a = a}}
           in f 1
```

$$f \mapsto \mathit{Lam}\ x\ (g\ (g\ x))$$
$$[g \mapsto \mathit{Lam}\ y\ (y + a)\ (a \mapsto \mathit{Dyn}\ \mathbf{int})]$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
⇝

let main = let f = let a = 5
                   in {g = {a = a}}
           in f' f 1

let f' (env: {g: {a: int}}) (x: int) =
  let g = env.g in g (g x)
```

$$f \mapsto Lam\ x\ (g\ (g\ x))$$
$$[g \mapsto Lam\ y\ (y + a)\ (a \mapsto Dyn\ \textbf{int})]$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
⇝


let main = let f = let a = 5
                   in {g = {a = a}}
           in f’ f 1

let f’ (env: {g: {a: int}}) (x: int) =
  let g = env.g in g (g x)
```

$$g \mapsto Lam\ y\ (y + a)\ [a \mapsto Dyn\ \mathbf{int}]$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
```

$\rightsquigarrow$

```
let main = let f = let a = 5
                   in {g = {a = a}}
           in f' f 1

let f' (env: {g: {a: int}}) (x: int) =
  let g = env.g in g' g (g' g x)

let g' (env: {a: int}) (y: int) =
  let a = env.a in y+a
```

$$g \mapsto \mathit{Lam}\ y\ (y + a)\ [a \mapsto \mathit{Dyn}\ \mathbf{int}]$$

Defunctionalization

```
let twice (g: int → int) = λx → g (g x)
let main = let f = let a = 5
                   in twice (λy → y+a)
           in f 1
⇝


let main = let f = let a = 5
                   in {g = {a = a}}
           in f' f 1

let f' (env: {g: {a: int}}) (x: int) =
  let g = env.g in g' g (g' g x)

let g' (env: {a: int}) (y: int) =
  let a = env.a in y+a
```

Correctness

Correctness

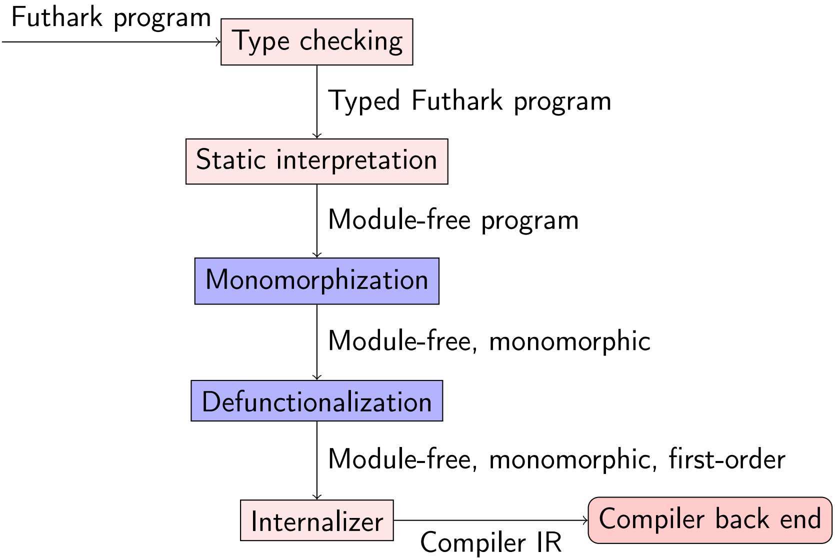
Defunctionalization has been proven correct:

- Defunctionalization terminates and yields a consistently typed residual expression.
  - For order 0, the type is unchanged.
  - Proof using a logical relations argument.
- Meaning is preserved.

More details in the paper.

Implementation

Implementation



Implementation

Polymorphism and defunctionalization

What if type a is instantiated with a function type?

```
let ite 'a (b: bool) (x: a) (y: a) : a =
  if b then x else y
```

Implementation

Polymorphism and defunctionalization

What if type a is instantiated with a function type?

```
let ite 'a (b: bool) (x: a) (y: a) : a =
  if b then x else y
```

Distinguish lifted type variables:

`'a`	regular type variable
`'^a`	*lifted* type variable

Evaluation

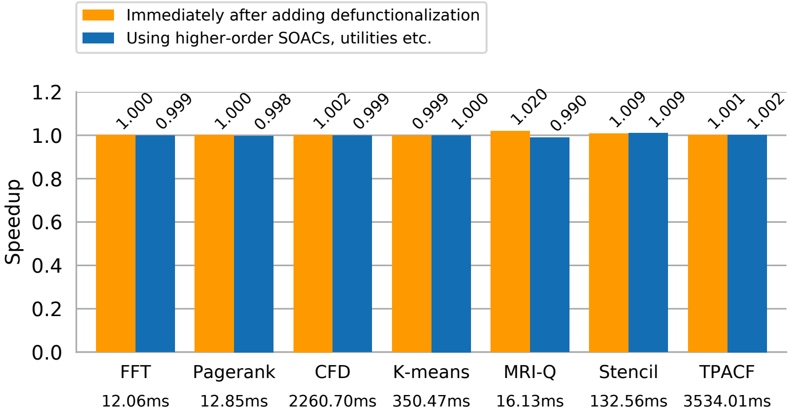
Evaluation

Does defunctionalization yield efficient programs?

Rewrite benchmark programs to use higher-order functions.

- Most SOACs converted to higher-order library functions.
- Higher-order utility functions
  - Function composition, application, `flip`, `curry`, etc.
- Segmented operations and sorting functions in library use higher-order functions instead of parametric modules.

Evaluation



- Run-time performance is unaffected.
- Relies on the optimizations performed by the compiler.

Functional images

Represent images as functions:

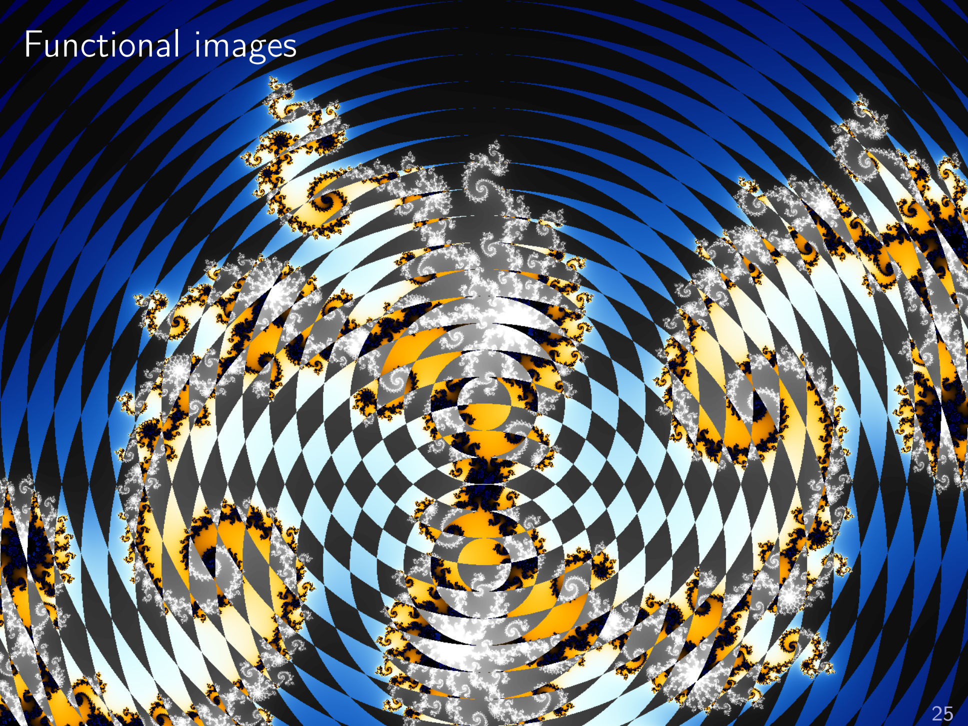
```
type image  'a = point → a

type filter 'a = image a → image a
```

- Due to Conal Elliott.
- Implemented in the Haskell EDSL *Pan*.

The entire Pan library has been translated to Futhark.

Functional images



Function-type conditionals

Support for function-type conditionals

```
let r = if b then {f = λx → x+1, a = 1}
             else {f = λx → x+n, a = 2}
in r.f r.a
```

Support for function-type conditionals

```
let r = if b then {f = λx → x+1, a = 1}
              else {f = λx → x+n, a = 2}
in r.f r.a
```

Introduce new form of static value:

$$Or\ sv_1\ sv_2$$

Static value representation of r:

$$\begin{aligned} Rcd\ \{f \mapsto\ & Or\ (Lam\ x\ (x+1)\ [\,])\\ &\quad\ (Lam\ x\ (x+n)\ [n \mapsto Dyn\ \mathbf{int}])\\ a \mapsto\ & Dyn\ \mathbf{int}\} \end{aligned}$$

Support for function-type conditionals

```
let r = if b then {f = λx → x+1, a = 1}
              else {f = λx → x+n, a = 2}
in r.f r.a
```

Straightforward translation is ill-typed:

```
⇝ if b then {f = {},    a = 1}
        else {f = {n=n}, a = 2}
```

Even worse with nested conditionals.

Binary sum types to complement *Or* static value:

$$\tau_1 + \tau_2$$

Support for function-type conditionals

```
let r = if b then {f = λx → x+1, a = 1}
             else {f = λx → x+n, a = 2}
in r.f r.a
```

$\rightsquigarrow$

```
let r = if b then {f = inl {},    a = 1}
             else {f = inr {n=n}, a = 2}
in let x = r.a
   in case r.f of
        inl e → x+1
        inr e → let n = e.n
                 in x+n
```

Conclusion

- General and practical approach to implementing higher-order functions in high-performance functional languages for GPUs.
- Proof of correctness.
- Implementation in Futhark.
- No performance overhead, but gain many of the benefits.

Questions, comments?