The Helium Haskell compiler

and its new LLVM backend

Ivo Gabe de Wolff - ivogabe@ivogabe.nl

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Haskell

- Functional
- Pure
- Lambda (function expression)
- Pattern matching
- Polymorphism
- Type classes (Traits in Rust, protocols in Swift)
- Lazy evaluation
- Partial application (currying)

Partial application

```haskell
divides :: Int -> Int -> Bool
divides a b = mod b a == 0

isEven :: Int -> Bool
isEven = divides 2
```

Desugared

```haskell
divides :: Int -> (Int -> Bool)
divides = \a -> (\b -> (mod b a) == 0)

isEven :: Int -> Bool
isEven = divides 2
```

Error messages: type graph

- Construct a graph containing type constraints
- Which constraints must be removed to make the graph consistent?

```
checks :: [Bool]
checks =
  [ divides 2
  , divides 3 5 ]
```

```
expression        : divides 2
term              : divides
  type            : Int -> Int -> Bool
  does not match  : Int -> Bool
because           : not enough arguments are given
```

Lazy evaluation

- Call-by-need semantics
- Thunk: object representing a computation
- Weak head normal form

Lazy evaluation

Sieve of Eratosthenes:

```
primes :: [Int]
primes = filterPrime [2..]
  where
    filterPrime (p:xs) =
      p : filterPrime (filter (\x -> not (divides p x)) xs)
```

Old backend: LVM

- Lazy Virtual Machine
- Stack-based instruction set
- Interpreted

Pipeline

- Haskell
- Core
- LVM

New backend: Iridium

- Strict, imperative language
- SSA
- Functional type system
- Pattern matching
- Laziness is explicit
- Multi-parameter functions

New backend: Iridium

```
export_as @null define @Prelude#null: { (forall a. ![a] -> Bool) }
  $ (forall v$2285, %u$0.434: ![v$2285]): Bool [trampoline] {
entry:
  case %u$0.434: ![v$2285] constructor (
    @"[]": (forall a. [a]) to case_nil,
    @":": (forall a. a -> [a] -> [a]) to case_cons)
case_nil:
  letalloc %.10378 = constructor @True: Bool $ ()
  return %.10378: !Bool
case_cons:
  letalloc %.10380 = constructor @False: Bool $ ()
  return %.10380: !Bool
}
```

Thunk

Object representing a computation or a partial application, containing:

- Pointer to a function or a thunk
- Number of given arguments
- Number of remaining arguments or a magic number
- Arguments

Evaluating a thunk

- Check if `remaining` is zero.
- Mark that the thunk is being evaluated by writing a magic number to `remaining`.
- Call the function pointer.
- Replace the function pointer by a pointer to the computed value.
- Write a magic number to `remaining`, indicating that the thunk is evaluated.

Pipeline

Core

1. Rename
2. Saturate
3. LetSort
4. LetInline
5. Normalize
6. Strictness
7. RemoveAliases
8. ReduceThunks
9. Lift

Iridium

1. ThunkArity
2. DeadCode
3. TailRecursion

Saturate - Correctness

Constructor applications should provide all arguments.

```
data Foo = Foo Int Bool String

x = Foo 1 True

x = \y -> Foo 1 True y
```

Let sorting - Optimization

Three kinds of *let* declarations: recursive, non-recursive and strict

```
let
  a = h b c
  b = f c
  c = g b
in [a, b, c]

let
  b = f c
  c = g b
in
  let a = h b c
  in [a, b, c]
```

LetInline - Optimization

Can we inline lazy let bindings?

```
let x = f 1
in g x x
```

```
g (f 1) (f 1)
```

- A thunk is evaluated at most once
- This may prevent inlining
- But some thunks are only used once

LetInline - Optimization

Inlines lazy non-recursive let bindings if one of the following holds:

- The definition of the variable is an unsaturated call
- The result of the thunk is not shared
- The variable is not used

Normalize - Correctness

Transform the program into a form where “most” subexpressions are variables.

```
x = f (g y)
```

```
x = let z = g y in f z
```

Strictness - Optimization

- Laziness is expensive and prevents other optimizations
- Analyze which expressions will always be used

```
x = let z = g y in f z

x = let! z = g y in f z
```

Strictness - Optimization

- Execution order unspecified
- Can change behavior when multiple expressions diverge

```haskell
error :: String -> a

x = error "A" + error "B"
```

RemoveAliases - Optimization

Removes aliasing of variables.

```
a = let x = y in f x
```

```
a = f y
```

```
b = let! x = y in
    let! z = x in f z
```

```
b = let! x = y in f x
```

ReduceThunks - Optimization

```
let a = 0 in f a

let! a = 0 in f a
```

Lift - Correctness

Transforms the program such that all lazy expressions are function or constructor applications.

Function expressions are lifted to toplevel declarations.

```
a = \x -> let y = expr in \z -> y + z
```

```
a = \x -> let y = b x in c x y
b = \x -> expr
c = \x -> \y -> \z -> y + z
```

Pipeline

Core

1. Rename
2. Saturate
3. LetSort
4. LetInline
5. Normalize
6. Strictness
7. RemoveAliases
8. ReduceThunks
9. Lift

Iridium

1. ThunkArity
2. DeadCode
3. TailRecursion

Iridium instructions

- Let - expressions such as call, phi, eval, literals
- LetAlloc - allocates thunks or constructors
- Jump
- Match - Extracts fields from an object
- Case - Conditional jump
- Return
- Unreachable

Iridium pipeline

- ThunkArity - Correctness
- DeadCode - Optimization
- TailRecursion - Optimization / correctness
- *Memory management*

Pipeline

Core

1. Rename
2. Saturate
3. LetSort
4. LetInline
5. Normalize
6. Strictness
7. RemoveAliases
8. ReduceThunks
9. Lift

Iridium

1. ThunkArity
2. DeadCode
3. TailRecursion

The Helium Haskell compiler

and its new LLVM backend

Ivo Gabe de Wolff - ivogabe@ivogabe.nl