Levity Polymorphism

Richard A. Eisenberg
Bryn Mawr College
rae@cs.brynmawr.edu

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

Tuesday, 20 June 2017
PLDI
Barcelona, Spain

How can we compile

polymorphism

without losing

performance

?

Polymorphism

parametric

```
choose :: ∀ a. Bool → a → a → a
choose True  t _ = t
choose False _ f = f

(+) :: ∀ a. Num a ⇒ a → a → a
```

"dictionary" of operations defined at type a

How can we compile polymorphism ?

Answer:
Many ways

Our novel approach:
kind-directed compilation

Design Criteria

- High performance
- Type erasure
- Support for fancy types
    - existential types
    - higher-rank types
    - polymorphic recursion

Compiling Polymorphism

- Uniform representation
  - Examples: Java, OCaml
  - All polymorphic values represented by pointers "boxed"
  - For OCaml: machine `int`s also work
  - Not performant

Compiling Polymorphism

- Uniform representation
- Monomorphization
  - Examples: C++, MLton, Rust
  - Polymorphic definitions are instantiated
  - No fancy types
  - Separate compilation is hard

Compiling Polymorphism

- Uniform representation
- Monomorphization
- Run-time specialization
  - C#: On-demand instantiation
  - TIL compiler for ML: runtime type analysis
  - No type erasure

Compiling Polymorphism

- Uniform representation
- Monomorphization
- Run-time specialization
- “Kinds are calling conventions”
  - Cyclone, TALT, Haskell/GHC

Kinds are calling conventions

```
choose :: Bool → a → a → a
```

```
let b = ... in
choose b 3 4
```

boxed ints

```
let b = ... in
choose b 3# 4#
```

machine ints

Kinds are calling conventions

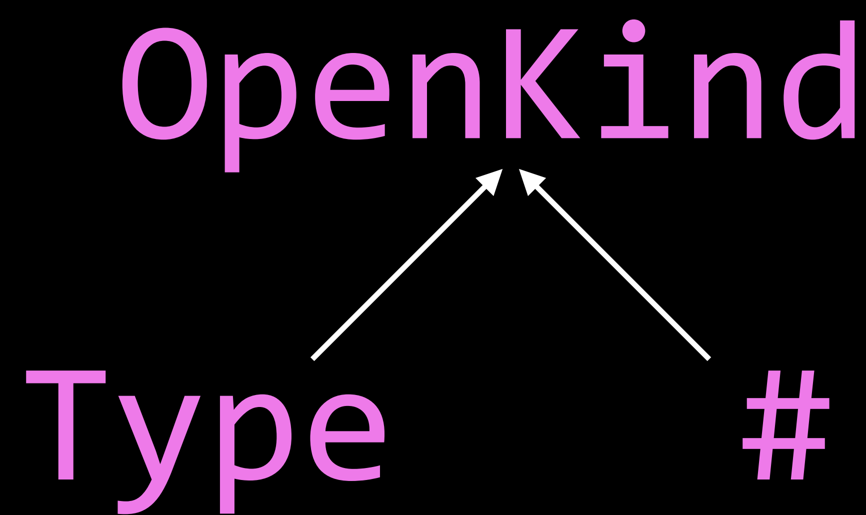
```
choose :: ∀ (a :: Type).
          Bool → a → a → a
 3 :: Int          3# :: Int#
 Int :: Type       Int# :: #

      let b = ... in
      choose b 3# 4#
```

kind mismatch

Problems lurk

- What is the kind of `(→)`?
  not `Type → Type → Type`
- Old solution: sub-kinding



- But that causes *more* problems

Our innovation:

Levity Polymorphism

Levity Polymorphism

```
TYPE :: Rep → Type
data Rep = LiftedRep
         | IntRep
         | DoubleRep
         | ...
type Type = TYPE LiftedRep
```

Examples

```
Int     :: Type
Int     :: TYPE LiftedRep
Int#    :: TYPE IntRep
Double# :: TYPE DoubleRep
Maybe   :: Type → Type
```

Examples

```
(+) :: ∀ (r :: Rep).
       ∀ (a :: TYPE r).
       Num a ⇒ a → a → a
```

3 + 4 3# + 4#

With levity polymorphism, performant code is easier to write.

Counter-Examples

```
choose :: ∀ (r :: Rep).
          ∀ (a :: TYPE r).
          Bool → a → a → a
choose True  t _ = t
choose False _ f = f
```

This cannot be compiled.

choose has to store its arguments.

Restrictions

Never store a levity-polymorphic value

- ➡ No levity-polymorphic variables
- ➡ No levity-polymorphic function arguments

GHC checks these

What can have L.P.?

```
($) :: ∀ (r :: Rep).
       ∀ (a :: Type)
         (b :: TYPE r).
       (a → b) → a → b
f $ x = f x
```

What can have L.P.?

```
error :: ∀ (r :: Rep)
            (a :: TYPE r).
          String → a
error msg = <throw exception>
```

What can have L.P.?

class methods

```
class Num (a :: TYPE r) where
  (+) :: a → a → a
  (-) :: a → a → a
  (*) :: a → a → a
  ...
```

34 of 76 standard classes can be generalized

What can have L.P.?

```
(→) ::
  ∀ (r1 :: Rep) (r2 :: Rep).
  TYPE r1 → TYPE r2 → Type
```

Kind-directed compilation

```
x = f y
```

How does GHC compile this function call?

Lazily or strictly?

It depends on the kind of the type of y.

The proof is in the paper.

Levity Polymorphism

Lazy types are lifted.
(They have an extra element.)

Levity polymorphism permits polymorphism over laziness, hence "liftedness".

Not liftedness, but levity.

With levity polymorphism, performant code is easier to write.

Levity Polymorphism

Richard A. Eisenberg
Bryn Mawr College
rae@cs.brynmawr.edu

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

Tuesday, 20 June 2017
PLDI
Barcelona, Spain