BRYN MAWR
COLLEGE

Stitch: The Sound Type-Indexed Type Checker

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

Wednesday, April 25, 2018
New York City Haskell Users' Group
New York, NY, USA

A brief history of Haskell types

- type classes (Wadler & Blott, POPL '89)
- functional dependencies (Jones, ESOP '00)
- data families (Chakravarty et al., POPL '05)
- type families (Chakravarty et al., ICFP '05)
- GADTs (Peyton Jones et al., ICFP '06)
- datatype promotion (Yorgey et al., TLDI '12)
- singletons (Eisenberg & Weirich, HS '12)
- `Type :: Type` (Weirich et al., ICFP '13)
- closed type families (Eisenberg et al., POPL '14)
- GADT pattern checking (Karachalias et al., ICFP '15)
- injective type families (Stolarek et al., HS '15)
- type application (Eisenberg et al., ESOP '16)
- new new `Typeable` (Peyton Jones et al., Wadlerfest '16)
- pattern synonyms (Pickering et al., HS '16)
- quantified class constraints (Bottu et al., HS '17)

How can we use all this technology?

Stitch!

```
> stitch
Welcome to the Stitch interpreter, version 1.0.
Type `:help` at the prompt for the list of commands.
λ> (\x:Int. x + 5) 3
8 : Int
λ> (\f:Int -> Int. \x:Int. f (f x)) (\x:Int. x + 5) 8
18 : Int
```

Download from:

https://cs.brynmawr.edu/~rae/pubs.html

(but you'll need GHC HEAD to compile)

Demo time!

De Bruijn indices

```
λ> \x:Int. \y:Int -> Int. y x
λ#:Int. λ#:Int -> Int. #0 #1 : Int -> (Int -> Int) -> Int
```

A de Bruijn index counts the number of intervening binders between a variable binding and its occurrence.

De Bruijn indices

Why?

- No shadowing
- Names are meaningless anyway
- Easier to formalize

Why not?

- Hard for humans

A type-indexed abstract syntax tree

```
data Exp :: forall n. Ctx n
         -> Type -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
  Lam :: TypeRep arg
      -> Exp (arg :> ctx) res
      -> Exp ctx (arg -> res)
  App :: Exp ctx (arg -> res)
      -> Exp ctx arg -> Exp ctx res
  ...
```

Language.Stitch.Exp

But first, we must parse!

A length-indexed abstract syntax tree

\# of vars in scope

```
data UExp (n :: Nat)
```

de Bruijn index

```
  = UVar (Fin n)
```

arg type

function body

```
  | ULam Ty (UExp (Succ n))
  | UApp (UExp n) (UExp n)
  | ULet (UExp n) (UExp (Succ n))
```

let-bound value

body

```
  | ...
```

Language.Stitch.Unchecked

What's that `Fin`?

`Fin` stands for finite set.

The type `Fin n` contains exactly `n` values.

What's that Fin?

```
data Nat = Zero | Succ Nat

data Fin :: Nat -> Type where
  FZ :: Fin (Succ n)
  FS :: Fin n -> Fin (Succ n)

              @2
    FS (FS FZ) :: Fin 5
              @0
    FS (FS FZ) :: Fin 3
             @???
    FS (FS FZ) :: Fin 2
```

Language.Stitch.Data.Fin

A length-indexed abstract syntax tree

```
data UExp (n :: Nat)
  = UVar (Fin n)
  | ULam Ty (UExp (Succ n))
  | UApp (UExp n) (UExp n)
  | ULet (UExp n) (UExp (Succ n))
  | ...
```

All variables must be well scoped

Language.Stitch.Unchecked

Well scoped parsing

How to parse an identifier?

```
~~var :: Parser (UExp n)~~
```

but we don't know
what n should be

To the code!

Types

Key idea:
use GHC's `TypeRep`

The value of type
`TypeRep a`
represents the type `a`.

Types

```
data TypeRep (a :: k)

class Typeable (a :: k)
```

produce a TypeRep

```
typeRep :: Typeable a => TypeRep a

eqTypeRep :: TypeRep a
          -> TypeRep b
          -> Maybe (a :~~: b)
```

compare TypeReps

Types

```
eqTypeRep :: TypeRep a
          -> TypeRep b
          -> Maybe (a :~~: b)

data (a :: k1) :~~: (b :: k2) where
  HRefl :: a :~~: a
```

heterogeneous (types have different kinds)
propositional (not automatically known by GHC)
equality

Types

```
eqTypeRep :: TypeRep a
          -> TypeRep b
          -> Maybe (a :~~: b)

data (a :: k1) :~~: (b :: k2) where
  HRefl :: a :~~: a
```

heterogeneous (types have different kinds)
propositional (not automatically known by GHC)
equality (two things that are the same)

Types

```
eqTypeRep :: TypeRep a
          -> TypeRep b
          -> Maybe (a :~~: b)

data (a :: k1) :~~: (b :: k2) where
  HRefl :: a :~~: a
```

pattern-matching HRefl
tells GHC that a ~ b.

```
cast :: a :~~: b -> a -> b
cast HRefl x = x
```

But first, we must parse!

Parsing a TypeRep

How to parse a TypeRep?

~~`ty :: Parser n (TypeRep t)`~~

but we don't know
what t should be

Existentials

```
data Ex :: (k -> Type) -> Type where
  Ex :: a i -> Ex a
```

i is existentially bound
(it is not mentioned in result type)

Thus, Ex TypeRep is a representation of any type.

```
type Ty = Ex (TypeRep :: Type -> Type)
```

A Ty represents a type of kind Type.

Parsing a TypeRep

How to parse a TypeRep?

```
ty :: Parser n Ty
```

```
data UExp (n :: Nat)
  = UVar (Fin n)
  | ULam Ty (UExp (Succ n))
  | UApp (UExp n) (UExp n)
  | ULet (UExp n) (UExp (Succ n))
  | ...
```

Milepost

- Parsed into a well scoped AST
- AST uses `Fin` for de Bruijn indices
- Parser indexed by # of vars in scope
- Parser env't is a length-indexed vec
- Parsing types requires existentials

A type-indexed abstract syntax tree

```
data Exp :: forall n. Ctx n
         -> Type -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
  Lam :: TypeRep arg
      -> Exp (arg :> ctx) res
      -> Exp ctx (arg -> res)
  App :: Exp ctx (arg -> res)
      -> Exp ctx arg -> Exp ctx res
  ...
```

Language.Stitch.Exp

A type-indexed abstract syntax tree

```
data Exp :: forall n. Ctx n
         -> Type -> Type
```

If

`exp :: Exp ctx ty`

then

$ctx \vdash exp : ty$

Language.Stitch.Exp

Contexts

```
type Ctx n = Vec Type n
```

yes, that Type

- A context is a vector of types.
- A de Bruijn index is just an index into this vector.

Contexts

```
type Ctx n = Vec Type n
```

yes, that Type

- A context is a vector of types.
- A de Bruijn index is just an index into this vector.

Language.Stitch.Exp

A type-indexed abstract syntax tree

CUSK

Polymorphic recursion

```
data Exp :: forall n. Ctx n
         -> Type -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
  Lam :: TypeRep arg
      -> Exp (arg :> ctx) res
      -> Exp ctx (arg -> res)
  App :: Exp ctx (arg -> res)
      -> Exp ctx arg -> Exp ctx res
  ...
```

Language.Stitch.Exp

A type-indexed abstract syntax tree

de Bruijn index

```
data Exp :: forall n. Ctx n
          -> Type -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
  Lam :: TypeRep arg
      -> Exp (arg :> ctx) res
      -> Exp ctx (arg -> res)
  App :: Exp ctx (arg -> res)
      -> Exp ctx arg -> Exp ctx res
  ...
```

Language.Stitch.Exp

Informative de Bruijn index

```
data Elem :: forall a n. Vec a n
          -> a -> Type where
  EZ :: Elem (x :> xs) x
```

x is either here…

```
  ES :: Elem xs x -> Elem (y :> xs) x
```

…or there

Language.Stitch.Data.Vec

Type checking

```
check :: UExp n -> M (Exp ctx ty)
```

Type checking

```
check :: UExp n -> M (Exp ctx ty)

check :: ∀ (ctx :: Ctx n).
         UExp n
      -> M (∃ ty. Exp ctx ty)
```

Type checking

```
check :: UExp n -> M (Exp ctx ty)
```

```
check :: ∀ (ctx :: Ctx n).
         UExp n
      -> M (∃ ty. Exp ctx ty)
```

```
check :: ∀ (ctx :: Ctx n).
         UExp n
      -> (∀ ty. Exp ctx ty -> M r)
      -> M r
```

Type checking

```
check :: ∀ (ctx :: Ctx n).
         UExp n
      -> (∀ ty. Exp ctx ty -> M r)
      -> M r
```

Type checking

```
check :: ∀ (ctx :: Ctx n).
           UExp n
        -> (∀ ty. Exp ctx ty -> M r)
        -> M r

check :: Sing (ctx :: Ctx n)
        -> UExp n
        -> (∀ ty. TypeRep ty
                   -> Exp ctx ty -> M r)
        -> M r
```

Type checking

```
check :: ∀ (ctx :: Ctx n).
         UExp n
      -> (∀ ty. Exp ctx ty -> M r)
      -> M r
```

```
check :: Sing (ctx :: Ctx n)
      -> UExp n
      -> (∀ ty. TypeRep ty
                -> Exp ctx ty -> M r)
      -> M r
```

Type checking

Yay -XTypeInType!

Singleton vector GADT

```
check :: Sing (ctx :: Ctx n)
      -> UExp n
      -> (∀ ty. TypeRep ty
             -> Exp ctx ty -> M r)
      -> M r
```

Language.Stitch.Check

To the code!

Evaluation

It's easy!

If it type-checks,

it works!

Common Subexpression Elimination

It's easy!

If it type-checks,

it works!

Common Subexpression Elimination

Generalized

```
data HashMap k v = ...
```

to

```
data IHashMap (k :: i -> Type)
              (v :: i -> Type) = ...
```

It took ~1hr for ~2k lines.

Common Subexpression Elimination

```
data IHashMap (k :: i -> Type)
              (v :: i -> Type) = ...
```

Writing instances requires quantified class constraints.

Conclusion

It's good to be fancy!

Dependent Types

- Stephanie Weirich and I have a grant
- *Lots* of GHC proposals
- Summer research students: Nadine, Dorothy, Eileen, My, Emma, Pablo, Ningning, and Matt
- Goals: merge type/term parsers, implement dependent Core, enable interactive error messages

Dependent Types

- Upcoming research leave: 2019-20
- Goal: Merge on π-day, 2021
- Help wanted!

Stitch: The Sound Type-Indexed Type Checker

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

https://cs.brynmawr.edu/~rae/pubs.html

Wednesday, April 25, 2018
New York City Haskell Users' Group
New York, NY, USA