tweag.io

BRYN MAWR COLLEGE

A Type-ical Case Study:

The Sound Type-Indexed Type Checker

Richard A. Eisenberg
Bryn Mawr College / Tweag I/O
rae@richarde.dev

Tarball at richarde.dev/stitch.tar.gz
and on ZuriHac website

Sunday, June 16, 2019
ZuriHac
Zürich, Switzerland

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)
- type abstractions (Eisenberg et al., HS '18)

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
```

Tarball at richarde.dev/stitch.tar.gz
and on ZuriHac website

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

Step 1: Lexing

boring, as usual

Language.Stitch.Lex

Step 2: Parsing

`Language.Stitch.Parse`

~~parseExp :: [LToken] -> UExp~~

errors, anyone?

~~parseExp :: [LToken]~~
~~-> Either String UExp~~

we want closed expressions

# of vars in scope

```
parseExp :: [LToken]
         -> Either String (UExp Zero)
```

A length-indexed abstract syntax tree

```
data Nat = Zero | Succ Nat

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

# of vars in scope

de Bruijn index

arg type

function body

let-bound value

body

Language.Stitch.Unchecked

What's that `Fin`?

`Fin` stands for finite set.

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

let's ignore laziness, shall we?

What's that Fin?

```
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

Parsing

```
parseExp :: [LToken]
         -> Either String (UExp Zero)
parseExp = ... expr ....
```

~~`expr :: Parser (UExp Zero)`~~

can't be recursive

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

n is only in output -- impossible

```
expr :: Parser n (UExp n)
```

Parsing

```
expr :: Parser n (UExp n)
type Parser n a
  -- a parser for an a with n vars in scope
  = ParsecT
      [LToken]     -- input
      ()           -- state
      (Reader (Vec String n)) -- monad
      a            -- result
```

var env

Language.Stitch.Parse

Vectors

```
data Vec :: Type -> Nat -> Type where
  VNil :: Vec a Zero
  (:>) :: a -> Vec a n
          -> Vec a (Succ n)
infixr 5 :>
```

A vec a n holds exactly
n elements of type a.

Language.Stitch.Data.Vec

Parsing

```
expr :: Parser n (UExp n)
type Parser n a
  -- a parser for an a with n vars in scope
  = ParsecT
      [LToken]          -- input
      ()                -- state
      (Reader (Vec String n)) -- monad
      a                 -- result
```

var env

Language.Stitch.Parse

To support well-scoped expressions, we need to index the parser monad and to use a length-indexed vector.

Types are social creatures.

Task: determine the collective noun for types.

(e.g., a closure of Haskellers)

Step 3: Type checking

```
data Ty = TInt
        | TBool
        | Ty :-> Ty
```

Language.Stitch.Type

A type-indexed abstract syntax tree

```
type Ctx n = Vec Ty n
data Exp :: forall n. Ctx n
          -> Ty -> Type where
```

Exp ctx ty is an expression of type ty in a context ctx.

If e :: Exp ctx ty, then ctx |- e : ty.

Language.Stitch.Exp

A type-indexed abstract syntax tree

```
type Ctx n = Vec Ty n
data Exp :: forall n. Ctx n
         -> Ty -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
```

de Bruijn index

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

"here"

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

"there"

Language.Stitch.Data.Vec

A type-indexed abstract syntax tree

```
type Ctx n = Vec Ty n
data Exp :: forall n. Ctx n
         -> Ty -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
  Lam :: STy arg
      -> Exp (arg :> ctx) res
      -> Exp ctx (arg :-> res)
```

Singleton (pointing to `STy arg`)

Need arg at compile time (indexing) and runtime (printing)

Language.Stitch.Exp

A type-indexed abstract syntax tree

```
Lam :: STy arg
    -> Exp (arg :> ctx) res
    -> Exp ctx (arg :-> res)
```

```
data STy :: Ty -> Type where
  SInt    :: STy TInt
  SBool   :: STy TBool
  (::->) :: STy arg -> STy res
          -> STy (arg :-> res)
```

Language.Stitch.Exp

A type-indexed abstract syntax tree

```
type Ctx n = Vec Ty n
data Exp :: forall n. Ctx n
            -> Ty -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
  Lam :: STy arg
      -> Exp (arg :> ctx) res
      -> Exp ctx (arg :-> res)
```


Language.Stitch.Exp

A type-indexed abstract syntax tree

```
type Ctx n = Vec Ty n
data Exp :: forall n. Ctx n
         -> Ty -> Type where
  Var :: Elem ctx ty -> Exp ctx ty
  Lam :: STy arg
      -> Exp (arg :> ctx) res
      -> Exp ctx (arg :-> res)
  App :: Exp ctx (arg :-> res)
      -> Exp ctx arg -> Exp ctx res
  ...
```

Language.Stitch.Exp

Type checking

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

what is ty?

```
check :: forall n (ctx :: Ctx n).
         UExp n
      -> M (exists ty. Exp ctx ty)
```

exists doesn't

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

Type checking

```
check not enough data at runtime
  :: forall n (ctx :: Ctx n) r.
     UExp n
  -> (forall ty. Exp ctx ty -> M r)
  -> M r
```

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

Type checking

singleton vector GADT

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

Language.Stitch.Check

To the code!

Step 4: 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.

Recap

- Identify a data invariant
- Check invariant with types
- Prove your code respects the invariant (using more types)
- Repeat

Conclusion

It's good to be fancy!

Dependent Types

- Grown to team effort!

Dependent Types

- Grown to team effort!

Code

Simon PJ

My Nguyen

Ryan Scott

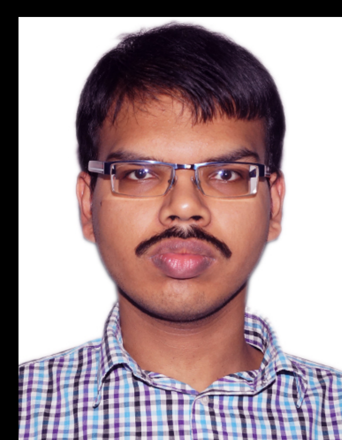
Vladislav Zavialov

Csongor Kiss

Ningning Xie

Stephanie Weirich

Antoine Voizard



Pritam Choudhury

Theory

Dependent Types

- Grown to team effort!
- Surprisingly, not really needed for Stitch
- Lots and lots of proposals: github.com/ghc-proposals/ghc-proposals/
- I will be working on GHC full-time this year, and will have more time for GHC for the foreseeable future (thanks to Tweag I/O)
- Join the fun! Commenting on proposals is a great way to start.
- Goal: Merge on π-day, 2021

tweag.io

BRYN MAWR COLLEGE

A Type-ical Case Study:

The Sound Type-Indexed Type Checker

Richard A. Eisenberg
Bryn Mawr College / Tweag I/O
rae@richarde.dev

Tarball at richarde.dev/stitch.tar.gz
and on ZuriHac website

Sunday, June 16, 2019
ZuriHac
Zürich, Switzerland