# Step 1: Lexing

boring, as usual

Language.Stitch.Lex

# Step 2: Parsing

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

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

errors, anyone?

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

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

errors, anyone?

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

we want <u>closed</u> expressions

\# of vars in scope

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

# A length-indexed abstract syntax tree

```
data Nat = Zero | Succ Nat
                              # 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.

let's ignore laziness, shall we?

# 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))
  | ...
```

$\boxed{\text{All variables must be well scoped}}$

Language.Stitch.Unchecked

# Parsing

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

expr :: Parser (UExp Zero)
```

# Parsing

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

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

can't be recursive

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

# 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
```

Language.Stitch.Parse

# 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

A vec a n stores exactly n as.

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.

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

"here"

"there"

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)
```

Singleton

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)
  App :: Exp ctx (arg :-> res)
      -> Exp ctx arg -> Exp ctx res
  ...
```

Language.Stitch.Exp

# Type checking

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

# 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)
```

# 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

not enough data

```
~~check
  :: 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!

TWEAG

# Stitch

## The Sound Type-Indexed Type Checker (Functional Pearl)

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

Tarball/repo linked from richarde.dev/pubs.html

Friday, 28 August 2020
Haskell Symposium

# 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

# 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