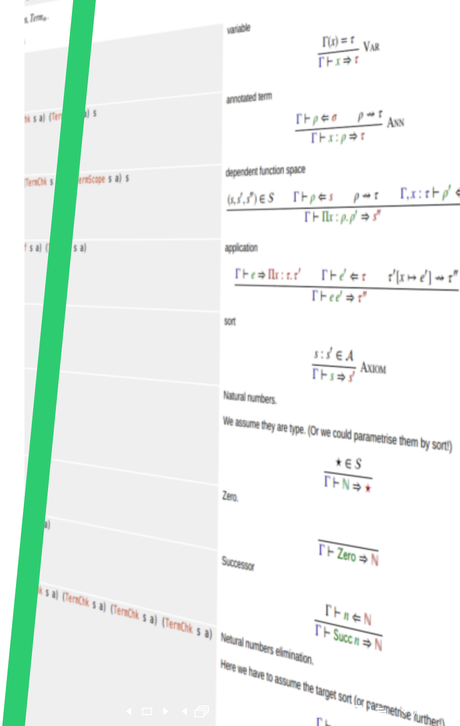
futurice

Write yourself a typed functional language

Oleg Grenrus

2018-10-11 • HASKELL EXCHANGE • LONDON



- Read a paper about type systems
- Implement a prototype
- ...
- Profit!

- Read a paper about type systems
- Implement a prototype
- ...
- Profit! = understand the paper

futurice

futurice

Fundamenta Informaticae XXI (2001) 1001–1031
IOS Press

A tutorial implementation of a dependently typed lambda calculus

Andres Löh
Utrecht University
andres@cs.uu.nl

Conor McBride
University of Strathclyde
conor.mcbride@cis.strath.ac.uk

Wouter Swierstra
University of Nottingham
wss@cs.nott.ac.uk

Abstract. We present the type rules for a dependently typed core calculus together with a straightforward implementation in Haskell. We explicitly highlight the changes necessary to shift from a simply-typed lambda calculus to the dependently typed lambda calculus. We also describe how to extend our core language with data types and write several small example programs. The article is accompanied by an executable interpreter and example code that allows immediate experimentation with the system we describe.

1. Introduction

Most functional programmers are hesitant to program with dependent types. It is said that type checking becomes undecidable; the type checker will always loop; and that dependent types are just really, really, hard.

Me

- Oleg Grenrus, `@phadej`

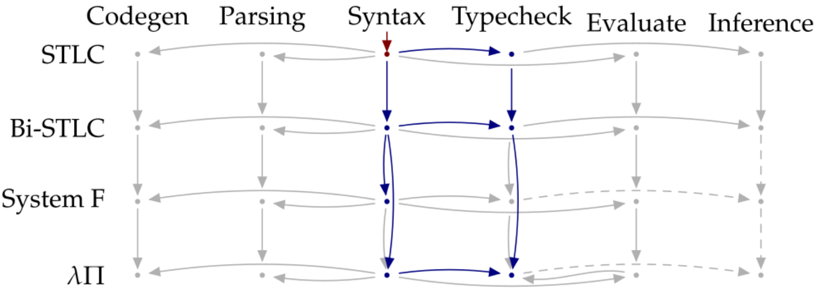
Me

- Oleg Grenrus, `@phadej`
- I work at futurice

Me

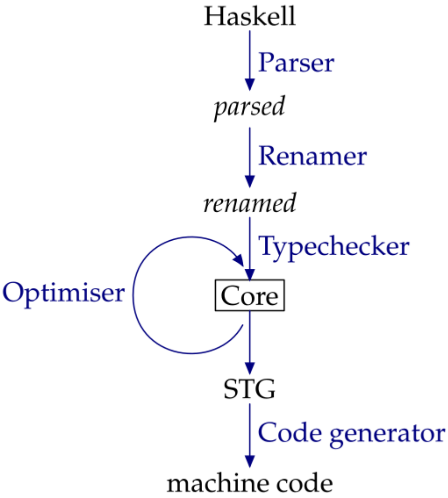
- Oleg Grenrus, `@phadej`
- I work at futurice
- I co-organise the HaskHel-meetup

- **Type-systems**: STLC, System F, dependent types…
- **"problems"**: representing syntax, type-checking, evaluation, type-inference…
- $n \times m$ combinations!

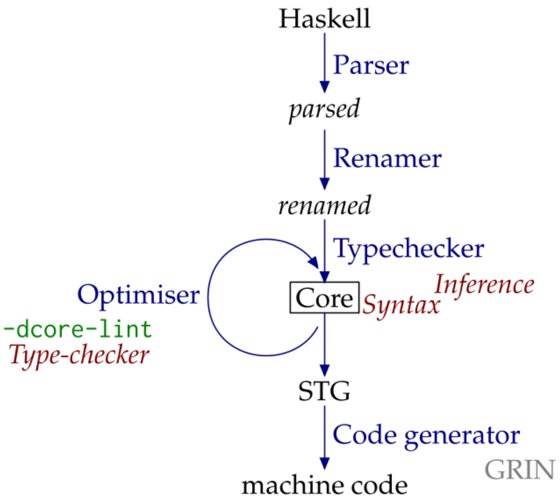

Codegen
Parsing
Syntax
Typecheck
Evaluate
Inference
STLC
Bi-STLC
System F
λΠ

futurice

Comparison to real compiler (= GHC)



Comparison to real compiler (= GHC)



Simple Typed Lambda Calculus

STLC

Term	
x	variable
$f\ x$	application
$\lambda(x : t) \to b$	abstraction

STLC

Term	
x	variable
$f\ x$	application
$\lambda(x : t) \to b$	abstraction

This is a typed language, so we annotate λ with an explicit type

STLC

Term	
x	variable
$f\ x$	application
$\lambda(x : t) \to b$	abstraction

$$
\begin{aligned}
\mathit{const}_{\mathsf{Int},\mathsf{Bool}} &\coloneqq \lambda(x : \mathsf{Int}) \to \lambda(y : \mathsf{Bool}) \to x \\
\mathit{apply}_{\mathsf{Char},\mathsf{Foo}} &\coloneqq \lambda(f : \mathsf{Char} \to \mathsf{Foo}) \to \lambda(x : \mathsf{Char}) \to f\ x \\
\mathit{example} &\coloneqq \mathit{mapIntInt}\ (\lambda(x : \mathsf{Int}) \to \mathit{sq}\ x)
\end{aligned}
$$

STLC

Term	
x	variable
$f\ x$	application
$\lambda(x : t) \to b$	abstraction

$$
\begin{aligned}
const_{\mathsf{Int},\mathsf{Bool}} &\coloneqq \lambda(x : \mathsf{Int}) \to \lambda(y : \mathsf{Bool}) \to x \\
apply_{\mathsf{Char},\mathsf{Foo}} &\coloneqq \lambda(f : \mathsf{Char} \to \mathsf{Foo}) \to \lambda(x : \mathsf{Char}) \to f\ x \\
example &\coloneqq mapIntInt\ (\lambda(x : \mathsf{Int}) \to sq\ x)
\end{aligned}
$$

Difficulties with λ terms

α-equivalence:

$\lambda(x : \tau) \to x \equiv \lambda(y : \tau) \to y$

Capture avoiding substitution:

$(\lambda(x : \tau) \to \lambda(y : \sigma) \to x)\ y \leadsto \lambda(z : \sigma) \to y \not\equiv \lambda(y : \sigma) \to y$

$a \leadsto b$: a reduces to b

futurice

How to represent names?

Term	
x	
$f\ x$	
$\lambda(x : t) \to b$	

```
data ExprText
  = Var Text
  | App Expr Expr
  | Lam Text Ty Expr
```

```
App (Var "mapIntInt") $
    Lam "x" (Ty "Int") $
        App (Var "sq") (Var "x")
```

Bad idea: both equivalence and substitution are difficult

Names: Raw Text

- α**-equivalence**: We need to maintain a mapping of names. Lambdas make that mapping trickier!
- **Capture avoiding substitution**: We need to rename terms, and also come up with fresh names.

Names: de Bruijn indices

Term	
c	`data ExprDeBruijn`
x	`= Free Text`
$f\ x$	`| Bound Natural` $\leftarrow$ index
$\lambda(x : t) \to b$	`| App Expr Expr`
	`| Lam Ty Expr`

```
App (Var "mapIntInt") $
    Lam (Ty "Int") $
        App (Var "sq") (Bound 0)
```

equivalence easy, substitution difficult

Names: de Bruijn indices

- α**-equivalence**: Structural equality, `Eq`
- **Capture avoiding substitution**: Some index juggling to get right

$$
\begin{aligned}
&\lambda(f : A \to A) \to (\lambda(x : A) \to \boxed{f}\ x)\ y \\
\equiv\ &\lambda\,(A \to A) \to (\lambda\,A \to \boxed{1}\ 0)\ y \\
\rightsquigarrow\ &\lambda\,(A \to A) \to \boxed{0}\ y \\
\equiv\ &\lambda\,(f : A \to A) \to \boxed{f}\ y
\end{aligned}
$$

Names: HOAS

Term	`data ExprHOAS`
c	`  = Free Text`
$f\ x$	`  | App Expr Expr`
$\lambda(x : t) \to b$	`  | Lam (Expr -> Expr)` $\longleftarrow$ Haskell function

```
App (Var "mapIntInt") $
    Lam (Ty "Int") $ \x ->
        App (Var "sq") x
```

equivalence difficult, substitution easy (and fast!)

Names: HOAS

futurice

- α**-equivalence**: cannot compare functions for equality!
- **Capture avoiding substitution**: "Outsourced" to the host language.

$$\lambda(f : A \to A) \to (\lambda(x : A) \to f\ x)\ y$$

$\equiv$ `Lam $ \f -> App (Lam $ \x -> f x) y`

$\rightsquigarrow$ `Lam $ \f -> f y`

$$\equiv \lambda\,(f : A \to A) \to f\ y$$

bound in one slide

"de Bruijn notation as a nested datatype" by Bird and Paterson, JFP99

`f ~ Expr`

```
abstract1 :: (Monad f, Eq a)
          => a -> f a -> Scope () f a

instantiate1 :: Monad f
             => f a -> Scope n f a -> f a

(>>>=) :: Monad f
       => Scope n f a
       -> (a -> f b)
       -> Scope n f b
```

`Scope n f a` is a `f a` with `n` holes of `f a` shape.

Names: bound

Term	
x	`data Expr a`
$f\ x$	`= Var a`
$\lambda(x : t) \to b$	`| App (Expr a) (Expr a)`
	`| Lam Ty (Scope () Expr a)`

```
App (Var "mapIntInt") $
  Lam (Ty "Int") $ abstract1 "x" $
    App (Var "sq") (Var "x")
```

equality easy (`Eq`), good enough for evaluation

futurice

STLC - Type-checking

Type-checking

```
typeCheck
    :: (a -> Maybe Ty)
    -> Expr a
    -> Maybe Ty
typeCheck ctx expr0 = do
    exprTy <- traverse ctx expr0
    go expr
  where
    go :: Expr Ty -> Maybe Ty
```

Working with bound version of `Expr`

Type-checking

```
typeCheck
    :: (a -> Maybe Ty)
    -> Expr a
    -> Maybe Ty
typeCheck ctx expr0 = do
    exprTy <- traverse ctx expr0
    go expr
  where
    go :: Expr Ty -> Maybe Ty
```

context: types of free variables

futurice

Type-checking

```
typeCheck
    :: (a -> Maybe Ty)
    -> Expr a
    -> Maybe Ty
typeCheck ctx expr0 = do
    exprTy <- traverse ctx expr0
    go expr
  where
    go :: Expr Ty -> Maybe Ty
```

`traverse` is an answer to many questions

futurice

Type-checking

```
typeCheck
    :: (a -> Maybe Ty)
    -> Expr a
    -> Maybe Ty
typeCheck ctx expr0 = do
    exprTy <- traverse ctx expr0
    go expr
  where
    go :: Expr Ty -> Maybe Ty
```

go is a recursive work horse

futurice

Type-checking: Var

$$\frac{}{\Gamma, x : A \vdash x : A}\ \text{Var}$$

```
go (Var ty) = return ty
```

As variables carry their / “are” types, we know their types

Type-checking: Lam

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A) \to e : A \to B}\ \text{Lam}$$

```
go (Lam a e) = do
    b <- go (instantiate1 (Var a) e)
    return (a :-> b)
```

Type-checking: Lam

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \boxed{\lambda(x : A) \to e} : A \to B}\ \text{Lam}$$

```
go (Lam a e) = do
    b <- go (instantiate1 (Var a) e)
    return (a :-> b)
```

We are checking the highlighted term

```
a :: Ty
e :: Scope () Expr a
```

Type-checking: Lam

$$\frac{\Gamma, \boxed{x : A \vdash e : B}}{\Gamma \vdash \lambda(x : A) \to e : A \to B}\ \text{Lam}$$

```
go (Lam a e) = do
    b <- go (instantiate1 (Var a) e)
    return (a :-> b)
```

Recursively checking the function body...

```
instantiate1 :: f a -> Scope n f a -> f a
```

futurice

Type-checking: Lam

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A) \to e : \boxed{A \to B}}\ \mathrm{Lam}$$

```
go (Lam a e) = do
    b <- go (instantiate1 (Var a) e)
    return (a :-> b)
```

And concluding with a function type

Type-checking: App

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B}\ \text{App}$$

```
go (App f x) = do
    ft <- go f
    case ft of
        a :-> b -> do
            xt <- go x
            guard (a == xt)
            return b
        _ -> fail "function type expected"
```

Type-checking: App

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash \boxed{f\ x} : B}\ \text{App}$$

```
go (App f x) = do
    ft <- go f
    case ft of
        a :-> b -> do
            xt <- go x
            guard (a == xt)
            return b
        _ -> fail "function type expected"
```

Type-checking: App

$$\frac{\Gamma \vdash \boxed{f} : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B}\ \text{App}$$

```
go (App f x) = do
    ft <- go f
    case ft of
        a :-> b -> do
            xt <- go x
            guard (a == xt)
            return b
        _ -> fail "function type expected"
```

futurice

Type-checking: App

$$\frac{\Gamma \vdash f : \boxed{A \to B} \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B}\ \text{App}$$

```
go (App f x) = do
    ft <- go f
    case ft of
        a :-> b  -> do
            xt <- go x
            guard (a == xt)
            return b
        _ -> fail "function type expected"
```

futurice

Type-checking: App

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash \boxed{x : A}}{\Gamma \vdash f\ x : B}\ \text{App}$$

```
go (App f x) = do
    ft <- go f
    case ft of
        a :-> b -> do
            xt <- go x
            guard (a == xt)
            return b
        _ -> fail "function type expected"
```

futurice

Type-checking: App

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : \boxed{B}}\ \text{App}$$

```
go (App f x) = do
    ft <- go f
    case ft of
        a :-> b -> do
            xt <- go x
            guard (a == xt)
            return b
        _ -> fail "function type expected"
```

Bidirectional type systems

Type annotations

$$\mathit{method} :: \mathsf{Foo} \rightarrow \mathsf{Bar} \rightarrow \mathsf{Quux}$$
$$\mathit{method}\ x\ y = \mathit{foo}\ x\ (\mathit{mkQuux}\ y :: \mathsf{Baz})$$

- In Haskell we like top-level type-signatures. Also in expressions (not so common).

Type annotations

$$\mathit{method} :: \mathsf{Foo} \to \mathsf{Bar} \to \mathsf{Quux}$$
$$\mathit{method}\ x\ y = \mathit{foo}\ x\ (\mathit{mkQuux}\ y :: \mathsf{Baz})$$

- In Haskell we like top-level type-signatures. Also in expressions (not so common).
- $$\begin{aligned} \mathit{method} &\coloneqq (\lambda(\mathtt{x} : \mathsf{Foo}) \to \lambda(\mathtt{y} : \mathsf{Bar}) \to \mathit{foo}\ \mathtt{x}\ (\mathit{mkQuux}\ \mathtt{y} : \mathsf{Baz})) \\ &: \ \mathsf{Foo} \to \mathsf{Bar} \to \mathsf{Quux} \end{aligned}$$

Can't we just add `Ann` constructor?

Term	
x	
$f\ x$	
$\lambda(x : t) \to b$	
$\boxed{x : t}$	

```haskell
data Expr a
  = Var a
  | App (Expr a) (Expr a)
  | Lam Ty (Scope () Expr a)
  | Ann (Expr a) Ty
```

Can't we just add `Ann` constructor?

Term	
x	`data Expr a`
$f\ x$	`  = Var a`
$\lambda(x : t) \to b$	`  | App (Expr a) (Expr a)`
$x : t$	`  | Lam Ty (Scope () Expr a)`
	`  | Ann (Expr a) Ty`

This works, but do we need a type annotation in `Lam`?

$$method \coloneqq (\lambda(x : \boxed{\mathsf{Foo}}) \to \lambda(y : \mathsf{Bar}) \to foo\ x\ (mkQuux\ y : \mathsf{Baz}))$$

$$: \boxed{\mathsf{Foo}} \to \mathsf{Bar} \to \mathsf{Quux}$$

Bidirectional STLC

Bidirectional type system requires less type annotations. Closer to an ideal surface language. Split `Expr` into two syntactic categories:

Synthed	`data Syn a`
x	`= Var a`
$s\ c$	`| App (Syn a) (Chk a)`
$c : t$	`| Ann (Chk a) Ty`

Checked	`data Chk a`
s	`= Syn (Syn a)`
$\lambda x \to c$	`| Lam (ScopeH () Chk Syn a)`

futurice

Bidirectional STLC

Bidirectional type system requires less type annotations. Closer to an ideal surface language. Split `Expr` into two syntactic categories:

Synthed	`data Syn a`
x	`= Var a`
$s\ c$	`| App (Syn a) (Chk a)`
$c : t$	`| Ann (Chk a) Ty`

Checked	`data Chk a`
s	`= Syn (Syn a)`
$\lambda x \to c$	`| Lam (ScopeH () Chk Syn a)`

futurice

Bidirectional STLC

Bidirectional type system requires less type annotations. Closer to an ideal surface language. Split `Expr` into two syntactic categories:

Synthed	`data Syn a`
x	`= Var a`
$s\ c$	`| App (Syn a) (Chk a)`
$c : t$	`| Ann (Chk a) Ty`

Checked	`data Chk a`
s	`= Syn (Syn a)`
$\lambda x \to c$	`| Lam (ScopeH () Chk Syn a)`

Bidirectional STLC

Bidirectional type system requires less type annotations. Closer to an ideal surface language. Split `Expr` into two syntactic categories:

Synthed	`data Syn a`
x	`= Var a`
$s\ c$	`| App (Syn a) (Chk a)`
$c : t$	`| Ann (Chk a) Ty`
Checked	`data Chk a`
s	`= Syn (Syn a)`
$\lambda x \to c$	`| Lam (ScopeH () Chk Syn a)`

futurice

Bidirectional STLC

Bidirectional type system requires less type annotations. Closer to an ideal surface language. Split `Expr` into two syntactic categories:

Synthed	`data Syn a`
x	`= Var a`
$s\ c$	`| App (Syn a) (Chk a)`
$c : t$	`| Ann (Chk a) Ty`

Checked	`data Chk a`
s	`= Syn (Syn a)`
$\lambda x \to c$	`| Lam (ScopeH () Chk Syn a)`
	`| Lam (Scope () Expr a)`

futurice

bound-extras in one slide

```
abstract1H :: (Functor f, Monad m, Eq a)
           => a -> f a -> ScopeH () f m a

instantiate1H :: Module f m
              => m a -> ScopeH b f m a -> f a

(>>==) :: Module f m
       => ScopeH n f m a
       -> (a -> m b)
       -> ScopeH n f m b
```

futurice

bound ↔ bound-extras

- `Scope n f a` is a `f a` with n holes of `f a` shape.
- `ScopeH n f m a` is a `f a` with n holes of `m a` shape.

bound ↔ bound-extras

- `Scope n f a` is a `f a` with n holes of $\boxed{\texttt{f a}}$ shape.
- `ScopeH n f m a` is a `f a` with n holes of $\boxed{\texttt{m a}}$ shape.

Type-checking

```
typeCheck :: (a -> Maybe Ty) -> Syn a -> Maybe Ty
typeCheck ctx expr0 = do
    exprTy <- traverse ctx expr0
    type_ exprTy
  where
    type_ :: Syn Ty -> Maybe Ty
    check_ :: Chk Ty -> Ty -> Maybe ()
```

futurice

Type-checking

```
typeCheck :: (a -> Maybe Ty) -> Syn a -> Maybe Ty
typeCheck ctx expr0 = do
    exprTy <- traverse ctx expr0
    type_ exprTy
  where
    type_ :: Syn Ty -> Maybe Ty
    check_ :: Chk Ty -> Ty -> Maybe ()
```

Same as with unidirectional STLC

Type-checking

```
typeCheck :: (a -> Maybe Ty) -> Syn a -> Maybe Ty
typeCheck ctx expr0 = do
    exprTy <- traverse ctx expr0
    type_ exprTy
  where
    type_ :: Syn Ty -> Maybe Ty
    check_ :: Chk Ty -> Ty -> Maybe ()
```

Two syntactic categories ⇒ two functions

futurice

Type synthesis: Var

$$\frac{}{\Gamma, x : A \vdash x \in A} \text{Var}$$

```
type_ (Var ty) = return ty
```

This is the same as in ordinary STLC

Type synthesis: App

$$\frac{\Gamma \vdash f \in A \to B \qquad \Gamma \vdash x \ni A}{\Gamma \vdash f\ x \in B}\ \text{App}$$

```
type_ (App f x) = do
  ft <- type_ f
  case ft of
    a :-> b -> do
      xt <- go x
      guard (a == xt)
      check_ x a
      return b
    _ -> fail "function type expected"
```

Type checking: Lam

$$\frac{\Gamma, x : A \vdash e \ni B}{\Gamma \vdash \lambda x \to e \ni A \to B}\ \text{Lam}$$

```
check_ (Lam  x) (a :-> b) =
     check_ (instantiate1H (Var a) x) b
 check_ (Lam _) _ = Nothing
```

Type checking: Lam

$$\frac{\Gamma, x : A \vdash e \ni B}{\Gamma \vdash \lambda x \to e \ni A \to B}\ \text{Lam}$$

```
check_ (Lam a x) (a :-> b) =
    b <- go (instantiate1 (Var a) x)
    return (a :-> b)
    check_ (instantiate1H (Var a) x) b
 check_ (Lam _) _ = Nothing
```

Type checking: Ann and Syn

$$\frac{\Gamma \vdash x \ni A}{\Gamma \vdash x : A \in A}\ \text{Ann} \qquad \frac{\Gamma \vdash x \in A}{\Gamma \vdash x \ni A}\ \text{Syn}$$

```
type_ (Ann x t) = do
   check_ x t
   return t

check_ :: Chk Ty -> Ty -> Maybe ()
check_ (Syn x) t = do
   t' <- type_ x
   guard (t == t')
```

Briefly on System F

System F

```
data Ty a
  = Ty a
  | Ty a :-> Ty a
  | Forall (Scope () Ty a)
```

Polymorphic types

•

System F

```
data Ty a
  = Ty a
  | Ty a :-> Ty a
  | Forall (Scope () Ty a)

data Syn b a = ...
  | AppTy (Syn b a) (Ty b)
data Chk b a = ...
  | LamTy (ScopeH (Flip Chk a) Ty b)
```

.

System F

```
data Ty a
  = Ty a
  | Ty a :-> Ty a
  | Forall (Scope () Ty a)

data Syn b a = ...
  | AppTy (Syn b a) (Ty b)
data Chk b a = ...
  | LamTy (ScopeH (Flip Chk a) Ty b)
```

Think `TypeApplications: foo @Int`

.

futurice

System F

```
data Ty a
  = Ty a
  | Ty a :-> Ty a
  | Forall (Scope () Ty a)

data Syn b a = ...
  | AppTy (Syn b a) (Ty b)
data Chk b a = ...
  | LamTy (ScopeH (Flip Chk a) Ty b)
```

Soon in GHC: `polymorphic @a = ...`

.

futurice

$\lambda\Pi$

λΠ

```
data Syn a
  = Var a
  | App (Syn a) (Chk a)
  | Ann (Chk a) Ty


data Chk a
  = Syn (Syn a)
  | Lam (ScopeH () Chk Syn a)
```

Let’s start with Bi-STLC

λΠ

```haskell
data Syn a
  = Var a
  | App (Syn a) (Chk a)
  | Ann (Chk a) (Syn a)


data Chk a
  = Syn (Syn a)
  | Lam (ScopeH () Chk Syn a)
```

Types are terms

λΠ

```
data Syn a
  = Var a
  | App (Syn a) (Chk a)
  | Ann (Chk a) (Syn a)
  | Syn a :-> Syn a

data Chk a
  = Syn (Syn a)
  | Lam (ScopeH () Chk Syn a)
```

A constructor for function type

λΠ

```
data Syn a
  = Var a
  | App (Syn a) (Chk a)
  | Ann (Chk a) (Syn a)
  | Pi (Syn a) (Scope () Syn a)

data Chk a
  = Syn (Syn a)
  | Lam (ScopeH () Chk Syn a)
```

$\Pi(x : A) \to B$ is a dependent function type, generalisation of $A \to B$ and $\forall A.B$

λΠ

```
data Syn a
  = Var a
  | App (Syn a) (Chk a)
  | Ann (Chk a) (Syn a)
  | Pi (Syn a) (Scope () Syn a)
  | Type
data Chk a
  = Syn (Syn a)
  | Lam (ScopeH () Chk Syn a)
```

$\mathbb{T}$ype is a type of types e.g. Int : $\mathbb{T}$ype,
but also including $\Pi(x : A) \to B$: $\mathbb{T}$ype and $\mathbb{T}$ype : $\mathbb{T}$ype

evalSyn :: Syn a -> Value a

- Even in System F types are naturally in **normal form**
- Dependent types are tricky: we need to normalise/evaluate them before comparing.
- Let's assume we have a function
  `evalSyn :: Syn a -> Value a`
  where `Value a` is a term in normal form.

Type-checking: Type

$$\frac{}{\Gamma \vdash \mathbb{T}\mathrm{ype} \in \mathbb{T}\mathrm{ype}}\ \text{Type}$$

```
type_ ctx Type = return VType
```

futurice

Type-checking: Type

$$\frac{}{\Gamma \vdash \mathbb{T}\text{ype} \in \mathbb{T}\text{ype}}\ \text{Type}$$

```
type_ ctx Type = return VType
```

We pass `ctx` for technical reasons.

Type-checking: Ann

$$\frac{\Gamma \vdash A \in \mathbb{T}\mathrm{ype} \qquad A \rightsquigarrow A' \qquad \Gamma \vdash x \ni A'}{\Gamma \vdash x : A \in A'}\ \text{Ann}$$

```
type_ ctx (Ann x a) = do
    at <- type_ ctx a
    guard (at == VType)
    let a' = evalSyn a
    check_ ctx x a'
    return a'
```

Type-checking: Ann

$$\frac{\Gamma \vdash A \in \mathbb{T}\mathsf{ype} \qquad A \rightsquigarrow A' \qquad \Gamma \vdash x \ni A'}{\Gamma \vdash \boxed{x : A} \in A'}\ \textsf{Ann}$$

```
type_ ctx (Ann x a) = do
    at <- type_ ctx a
    guard (at == VType)
    let a' = evalSyn a
    check_ ctx x a'
    return a'
```

Type-checking: Ann

$$\frac{\Gamma \vdash \boxed{A \in \mathbb{T}\mathrm{ype}} \qquad A \rightsquigarrow A' \qquad \Gamma \vdash x \ni A'}{\Gamma \vdash x : A \in A'}\ \text{Ann}$$

```
type_ ctx (Ann x a) = do
    at <- type_ ctx a
    guard (at == VType)
    let a' = evalSyn a
    check_ ctx x a'
    return a'
```

Type-checking: Ann

$$\frac{\Gamma \vdash A \in \mathbb{T}\mathsf{ype} \qquad \boxed{A \rightsquigarrow A'} \qquad \Gamma \vdash x \ni A'}{\Gamma \vdash x : A \in A'}\ \textsf{Ann}$$

```
type_ ctx (Ann x a) = do
    at <- type_ ctx a
    guard (at == VType)
    let a' = evalSyn a
    check_ ctx x a'
    return a'
```

Type-checking: Ann

$$\frac{\Gamma \vdash A \in \mathbb{T}\mathsf{ype} \qquad A \rightsquigarrow A' \qquad \Gamma \vdash \boxed{x \ni A'}}{\Gamma \vdash x : A \in A'}\ \mathsf{Ann}$$

```
type_ ctx (Ann x a) = do
    at <- type_ ctx a
    guard (at == VType)
    let a' = evalSyn a
    check_ ctx x a'
    return a'
```

Type-checking: Ann

$$\frac{\Gamma \vdash A \in \mathbb{T}\mathrm{ype} \qquad A \rightsquigarrow A' \qquad \Gamma \vdash x \ni A'}{\Gamma \vdash x : A \in \boxed{A'}}\ \text{Ann}$$

```
type_ ctx (Ann x a) = do
    at <- type_ ctx a
    guard (at == VType)
    let a' = evalSyn a
    check_ ctx x a'
    return a'
```

Type-checking: App

$$\frac{\Gamma \vdash f \in \Pi(y : A) \to B \qquad \Gamma \vdash x \ni A \qquad B[y \mapsto x] \leadsto B'}{\Gamma \vdash f\ x \in B'}\ \text{App}$$

```
type_ ctx (App f x) = do
    f' <- type_ ctx f
    case f' of
        VPi a b -> do
            check_ ctx x a
            let b' = S.instantiate1 (evalChk x) b
            return b'
        _ -> fail "Pi type expected"
```

Type-checking: App

$$\frac{\Gamma \vdash f \in \Pi(y : A) \to B \qquad \Gamma \vdash x \ni A \qquad B[y \mapsto x] \rightsquigarrow B'}{\Gamma \vdash \boxed{f\ x} \in B'}\ \text{App}$$

```
type_ ctx (App f x) = do
    f' <- type_ ctx f
    case f' of
        VPi a b -> do
            check_ ctx x a
            let b' = S.instantiate1 (evalChk x) b
            return b'
        _ -> fail "Pi type expected"
```

futurice

Type-checking: App

$$\frac{\Gamma \vdash \boxed{f \in \Pi(y : A) \to B} \qquad \Gamma \vdash x \ni A \qquad B[y \mapsto x] \rightsquigarrow B'}{\Gamma \vdash f\ x \in B'}\ \text{App}$$

```
type_ ctx (App f x) = do
    f' <- type_ ctx f
    case f' of
        VPi a b -> do
            check_ ctx x a
            let b' = S.instantiate1 (evalChk x) b
            return b'
        _ -> fail "Pi type expected"
```

Type-checking: App

$$\frac{\Gamma \vdash f \in \Pi(y : A) \to B \qquad \boxed{\Gamma \vdash x \ni A} \qquad B[y \mapsto x] \rightsquigarrow B'}{\Gamma \vdash f\ x \in B'}\ \text{App}$$

```
type_ ctx (App f x) = do
    f' <- type_ ctx f
    case f' of
        VPi a b -> do
            check_ ctx x a
            let b' = S.instantiate1 (evalChk x) b
            return b'
        _ -> fail "Pi type expected"
```

Type-checking: App

$$\frac{\Gamma \vdash f \in \Pi(y : A) \to B \qquad \Gamma \vdash x \ni A \qquad \boxed{B[y \mapsto x] \rightsquigarrow B'}}{\Gamma \vdash f\ x \in B'}\ \text{App}$$

```
type_ ctx (App f x) = do
    f' <- type_ ctx f
    case f' of
        VPi a b -> do
            check_ ctx x a
            let b' = S.instantiate1 (evalChk x) b
            return b'
        _ -> fail "Pi type expected"
```

Type-checking: App

$$\frac{\Gamma \vdash f \in \Pi(y : A) \to B \qquad \Gamma \vdash x \ni A \qquad B[y \mapsto x] \rightsquigarrow B'}{\Gamma \vdash f\ x \in \boxed{B'}}\ \text{App}$$

```
type_ ctx (App f x) = do
    f' <- type_ ctx f
    case f' of
        VPi a b -> do
            check_ ctx x a
            let b' = S.instantiate1 (evalChk x) b
            return b'
        _ -> fail "Pi type expected"
```

Conclusion

- We went through STLC, bidirectional type systems, dependent types
- `bound(-extras)` handle technical bits for us
- Typing rules are itself a very consice language (we hopefully understand now)

futurice

futurice

Thank you!
Questions?

Oleg Grenrus

@phadej

github.com/phadej/language-pts

hackage.haskell.org/package/bound-extras

Extra slides

Linear types!

- You can do linear types too!
- rules are more complicated, so is the code.

$$\frac{\Delta \vdash f : A \multimap B; \Delta' \qquad \Delta' \vdash x : A; \Delta''}{\Delta \vdash f\ x : B; \Delta''}\ \text{Lin-App}$$

```haskell
typeCheck :: Eq a ⇒ (a → Maybe Ty) → Expr a → Maybe Ty
typeCheck ctx expr = evalStateT (go expr) ctx

go :: ∀a.Eq a ⇒ Expr a → StateT (a → Maybe Ty) Maybe Ty
go (Var x) = do
  ctx ← get
  case ctx x of
    Nothing → fail "unbound variable"
    Just ty → do
      put (λy → if x ≡ y then Nothing else ctx y)
      return ty
go (App f x) = do
  ft ← go f
  case ft of
    (a :→ b) → do
      xt ← go x
      guard (a ≡ xt)
      return b
    _ → fail "Function type expected"
go (Lam a x) =
  let x' :: Expr (Var () a)
      x' = fromScope x
  in StateT $ λctx0 → do
    let ctx1 :: Var () a → Maybe Ty
        ctx1 = unvar (const (Just a)) ctx0
    (b, ctx2) ← runStateT (go x') ctx1
    case ctx2 (B ()) of
      Nothing → do
        let ctx3 :: a → Maybe Ty
            ctx3 = ctx2 ∘ F
        return (a :→ b, ctx3)
      Just _ → fail "non consumed"
```

Typechecker may elaborate!

For example from bidirectional to unidirectional:

$$toSTLC :: (a \to \mathsf{Maybe\ Ty}) \to \mathsf{Syn}\ a \to \mathsf{Maybe}\ (\mathsf{Ty}, \mathsf{Uni.Expr}\ a)$$

```
toSTLC :: (a → Maybe Ty) → Syn a → Maybe (Ty, Uni.Expr a)
toSTLC = type_ where
  type_  :: (a → Maybe Ty) → Syn a → Maybe (Ty, Uni.Expr a)
  check_ :: (a → Maybe Ty) → Chk a → Ty → Maybe (Uni.Expr a)

  type_ ctx (Var x) = do
    ty ← ctx x
    return (ty, Uni.Var x)

  type_ ctx (App f x) = do
    (ft, f′) ← type_ ctx f
    case ft of
      (a :→ b) → do
        x′ ← check_ ctx x a
        return (b, Uni.App f′ x′)
      _ → Nothing
```

...

… non-written things!

Using `unification-fd` library, we can write simple type inference:

$$
\begin{aligned}
&\mathit{infer}\ (\mathsf{Var}\ (\mathit{ty}, a)) = \mathsf{do} \\
&\quad \mathit{pure}\ (\mathsf{V}\ a, \mathit{ty}) \\
&\mathit{infer}\ (\mathsf{App}\ f\ x) = \mathsf{do} \\
&\quad (x', a) \leftarrow \mathit{infer}\ x \\
&\quad (f', ab) \leftarrow \mathit{infer}\ f \\
&\quad \mathtt{case}\ ab\ \mathtt{of} \\
&\quad\quad \mathsf{UTerm}\ (a' :\Rightarrow b') \rightarrow \mathsf{do} \\
&\quad\quad\quad \mathit{unify}\ a'\ a \quad \text{-- not } \mathit{guard}, \mathit{unify} \\
&\quad\quad\quad \mathit{return}\ (\mathsf{App}\ f'\ x', b') \\
&\ldots
\end{aligned}
$$