Intro to Type Systems and Operational Semantics

ZuriHac 2022

Andres Löh

2022-06-12 —

Well-Typed
The Haskell Consultants

About Well-Typed

- Well-Typed is a Haskell consultancy company, established in 2008
- Team of about 20 Haskell experts
- Wide variety of clients

- GHC and tooling maintenance, development and support
- Haskell software development and consulting
- On-site and remote training courses

About me

- Using Haskell since about 1997
- Studied mathematics in Konstanz, PhD in Computer Science at Utrecht 2004
- At Well-Typed since 2010
- Living in Regensburg, Germany

Credit

Most of the material in this lecture is inspired by

Benjamin Pierce
Types and Programming Languages

(but there are changes and all mistakes are my own).

There are many other good books / tutorials / blog posts.

Plan

Look at a succession of languages with more and more features:

- discuss type rules and how to read / write them,
- discuss evaluation / operational semantics,
- discuss a few properties,
- discuss a straight-forward implementation in Haskell.

Languages:

- a very simple language for expressions with Booleans and naturals,
- adding variables,
- adding functions (Simply Typed Lambda Calculus),
- adding polymorphism (System F).

What we will not do

There are many interesting things we will not have time to cover:

- type inference,
- type classes,
- kinds,
- an efficient implementation,
- a particularly type-safe implementation,
- ...

Code

All the code of this presentation (and the slides, and some exercises) are available from:

`https://github.com/well-typed/types-zurihac-2022`

The first steps (Version 1)

Natural numbers

Haskell:

```
data Nat = Z | S Nat
```

Syntax:

```
n ::= Z
   |  S n
```

Type rules:

$$\text{Z}\ \frac{}{\mathtt{Z : Nat}} \qquad\qquad \text{S}\ \frac{\mathtt{n : Nat}}{\mathtt{S\ n : Nat}}$$

Three is a natural number

$$\mathsf{S}\ \dfrac{\mathsf{S}\ \dfrac{\mathsf{S}\ \dfrac{\mathsf{Z}\ \dfrac{}{\texttt{Z : Nat}}}{\texttt{S Z : Nat}}}{\texttt{S (S Z) : Nat}}}{\texttt{S (S (S Z)) : Nat}}$$

Three is a natural number

$$\mathrm{S}\;\dfrac{\mathrm{S}\;\dfrac{\mathrm{S}\;\dfrac{\mathrm{Z}\;\dfrac{}{\texttt{Z : Nat}}}{\texttt{S Z : Nat}}}{\texttt{S (S Z) : Nat}}}{\texttt{S (S (S Z)) : Nat}}$$

Boring: All syntactically correct terms are also well-typed.

A larger language

Haskell:

```haskell
data Expr =
      Z
    | S     Expr
    | F
    | T
    | Equal Expr Expr
    | If    Expr Expr Expr
```

Syntax:

$$
\begin{aligned}
e ::= \; & \mathtt{Z} \\
\mid \; & \mathtt{S}\ e \\
\mid \; & \mathtt{False} \\
\mid \; & \mathtt{True} \\
\mid \; & e_1 \mathrel{==} e_2 \\
\mid \; & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3
\end{aligned}
$$

Well-Typed

Type rules

$$\text{Z}\ \frac{}{\texttt{Z : Nat}} \qquad \text{S}\ \frac{\texttt{e : Nat}}{\texttt{S e : Nat}}$$

$$\text{F}\ \frac{}{\texttt{F : Bool}} \qquad \text{T}\ \frac{}{\texttt{T : Bool}}$$

$$\text{Equal}\ \frac{\texttt{e}_1\ \texttt{: Nat} \qquad \texttt{e}_2\ \texttt{: Nat}}{\texttt{e}_1\ \texttt{==}\ \texttt{e}_2\ \texttt{: Bool}}$$

$$\text{If}\ \frac{\texttt{e}_1\ \texttt{: Bool} \qquad \texttt{e}_2\ \texttt{: t} \qquad \texttt{e}_3\ \texttt{: t}}{\textbf{if}\ \texttt{e}_1\ \textbf{then}\ \texttt{e}_2\ \textbf{else}\ \texttt{e}_3\ \texttt{: t}}$$

Example

$$\text{If}\;\dfrac{\text{Equal}\;\dfrac{\text{Z}\;\dfrac{}{\texttt{Z : Nat}} \qquad \text{S}\;\dfrac{\text{Z}\;\dfrac{}{\texttt{Z : Nat}}}{\texttt{S Z : Nat}}}{\texttt{Z == S Z : Bool}} \qquad \text{S}\;\dfrac{\text{S}\;\dfrac{\text{Z}\;\dfrac{}{\texttt{Z : Nat}}}{\texttt{S Z : Nat}}}{\texttt{S (S Z) : Nat}} \qquad \text{Z}\;\dfrac{}{\texttt{Z : Nat}}}{\texttt{if Z == S Z then S (S Z) else Z : Nat}}$$

Language of types

Syntax:

```
t ::= Nat | Bool
```

Haskell:

```haskell
data Ty = TNat | TBool
  deriving Eq
```

Language of types

Syntax:

```
t ::= Nat | Bool
```

Haskell:

```haskell
data Ty = TNat | TBool
  deriving Eq
```

Do we need types of types?

An implementation

```haskell
check :: Expr -> Ty -> Maybe ()
check e t = do
  t' <- infer e
  guard (t == t')
```

```haskell
infer :: Expr -> Maybe Ty
```

Inference

```
infer :: Expr -> Maybe Ty
infer Z     = pure TNat
infer (S e) = do
  check e TNat
  pure TNat
...
```

Inference

```
...
infer F               = pure TBool
infer T               = pure TBool
infer (Equal e₁ e₂) = do
  check e₁ TNat
  check e₂ TNat
  pure TBool
infer (If e₁ e₂ e₃) = do
  check e₁ TBool
  t <- infer e₂
  check e₃ t
  pure t
```

Evaluation

Values

What are the results of evaluation?

The answer depends to some extent on the evaluation strategy.

Values

What are the results of evaluation?

The answer depends to some extent on the evaluation strategy.

Values (normal forms, eager evaluation):

```
v ::= Z | S v | F | T
```

Values are a subset of expressions.

Values

What are the results of evaluation?

The answer depends to some extent on the evaluation strategy.

Values (normal forms, eager evaluation):

```
v ::= Z | S v | F | T
```

Values are a subset of expressions.

For call-by-name, or lazy evaluation, weak head normal forms are more interesting:

```
w ::= Z | S e | F | T
```

Also a subset of expressions.

Examples

```
S (S Z)
```

is in normal form (and weak head normal form).

```
S (if T then Z else S Z)
```

is in weak head normal form, but not in normal form.

Haskell values

```
data NF = VNat Nat | VBool Bool
```

```
data WHNF = WNat WNat | WBool Bool
data WNat = WZ | WS Expr
```

Well-Typed

Evaluation

```
eval :: Expr -> Maybe WHNF
eval Z              = pure (WNat WZ)
eval (S e)          = pure (WNat (WS e))
eval F              = pure (WBool False)
eval T              = pure (WBool True)
eval (Equal e₁ e₂) = do
  WNat v₁ <- eval e₁
  WNat v₂ <- eval e₂
  case (v₁, v₂) of
    (WZ, WZ) -> pure (WBool True)
    (WS e₁', WS e₂') -> eval (Equal e₁' e₂')
    _ -> pure (WBool False)
eval (If e₁ e₂ e₃) = do
  WBool b <- eval e₁
  if b then eval e₂ else eval e₃
```

Well-Typed

Evaluation rules

Two common styles

Big-step semantics:

$$\texttt{e} \Downarrow \texttt{w}$$

Relates an expression and a value.

Expectation: Values evaluate to themselves.

Two common styles

Big-step semantics:

$$\texttt{e} \Downarrow \texttt{w}$$

Relates an expression and a value.

Expectation: Values evaluate to themselves.

Small-step semantics:

$$\texttt{e} \longrightarrow \texttt{e'}$$

Relates two expressions.

Expectation: Values do not evaluate further.

Big-step evaluation rules

Evaluating data constructors:

$$\text{e-Z}\quad \frac{}{\texttt{Z} \Downarrow \texttt{Z}} \qquad\qquad \text{e-S}\quad \frac{}{\texttt{S e} \Downarrow \texttt{S e}}$$

$$\text{e-False}\quad \frac{}{\texttt{False} \Downarrow \texttt{False}} \qquad\qquad \text{e-True}\quad \frac{}{\texttt{True} \Downarrow \texttt{True}}$$

Evaluating equality

$$\text{e-Equal-Z-Z}\quad \frac{e_1 \Downarrow \mathtt{Z} \qquad e_2 \Downarrow \mathtt{Z}}{e_1 \mathbin{\mathtt{==}} e_2 \Downarrow \mathtt{True}}$$

$$\text{e-Equal-S-S}\quad \frac{e_1 \Downarrow \mathtt{S}\ e_1' \qquad e_2 \Downarrow \mathtt{S}\ e_2' \qquad e_1' \mathbin{\mathtt{==}} e_2' \Downarrow v}{e_1 \mathbin{\mathtt{==}} e_2 \Downarrow v}$$

$$\text{e-Equal-Z-S}\quad \frac{e_1 \Downarrow \mathtt{Z} \qquad e_2 \Downarrow \mathtt{S}\ e_2'}{e_1 \mathbin{\mathtt{==}} e_2 \Downarrow \mathtt{False}}$$

$$\text{e-Equal-S-Z}\quad \frac{e_1 \Downarrow \mathtt{S}\ e_1' \qquad e_2 \Downarrow \mathtt{Z}}{e_1 \mathbin{\mathtt{==}} e_2 \Downarrow \mathtt{False}}$$

Evaluating if-then-else

$$\text{e-If-False}\ \frac{e_1 \Downarrow \mathtt{False} \qquad e_3 \Downarrow v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v}$$

$$\text{e-If-True}\ \frac{e_1 \Downarrow \mathtt{True} \qquad e_2 \Downarrow v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v}$$

Small-step evaluation rules

No rules for data constructors needed.

Equality:

s-Equal-Z-Z `Z == Z` $\longrightarrow$ `True`

s-Equal-S-S `S` e_1 `==` `S` e_2 $\longrightarrow$ e_1 `==` e_2

s-Equal-Z-S `Z == S` e $\longrightarrow$ `False`

s-Equal-S-Z `S` e `== Z` $\longrightarrow$ `False`

Small-step if-then-else

s-If-False $\texttt{if False then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2$

s-If-True $\texttt{if True then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1$

Context rules

$$\text{s-Equal-1}\quad \frac{e_1 \longrightarrow e_1'}{e_1 \;\texttt{==}\; e_2 \longrightarrow e_1' \;\texttt{==}\; e_2}$$

$$\text{s-Equal-2}\quad \frac{e_2 \longrightarrow e_2'}{v_1 \;\texttt{==}\; e_2 \longrightarrow v_1 \;\texttt{==}\; e_2'}$$

$$\text{s-If-1}\quad \frac{e_1 \longrightarrow e_1'}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

Well-Typed

What is better?

There is no clear answer:

- The main disadvantage of small-step semantics is the need for context rules.
- On the other hand, the non-context rules are often a bit simpler.
- For big-step semantics, it is easier to identify the values.
- The main disadvantage of big-step semantics is that it is not easy to distinguish stuck terms from non-terminating terms.

Properties of type systems

Preservation / subject reduction

When we reduce an expression, we expect the type to be maintained.

For big-step semantics:

If `e : t` and $e \Downarrow v$, then `v : t`.

For small-step semantics:

If `e : t` and $e \longrightarrow e'$, then `e' : t`.

Progress

"Well-typed programs do not go wrong."

When an expression is well-typed, we expect it not to get stuck.

For small-step semantics:

If `e : t` , then there exists an `e'` such that `e ⟶ e'` .

Not easy to express for big-step semantics (unless all terms are terminating).

Adding variables (Version 2)

Expressions with let

```
e ::= ...
    |  let x = e₁ in e₂
    |  x
```

Adds a whole new class of problems:

- Where is `x` in scope? (Also: is `let` recursive?)
- What if we refer to a variable outside of its scope?
- What about name shadowing / name capture?
- Substitution?

Contextual information

What does `x == y` reduce to?

What is the type of `if True then x else x` ?

Environments / contexts

Environments are finite maps providing information about the types (or implementation of variables).

Syntax (for type environments):

$$\begin{array}{lcl} \Gamma & ::= & \varepsilon \\ & \mid & \Gamma, x : t \end{array}$$

Membership (for type environments):

$$\text{env-elem-1} \quad \overline{x : t \in \Gamma, x : t}$$

$$\text{env-elem-2} \quad \frac{x_1 \neq x_2 \qquad x : t_1 \in \Gamma}{x_1 : t_1 \in \Gamma, x_2 : t_2}$$

Well-Typed

Type-checking with a context

$$\Gamma \vdash e : a$$

Note that this is just syntax for a relation between three entities.

Most rules just ignore the context

$$\text{If}\ \frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : t}$$

Handling variables and let

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

$$\text{Let}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : t_2}$$

Note how even the type rule makes it clear that our `let` is not recursive.

Evaluation

s-let $\texttt{let } x = e_1 \texttt{ in } e_2 \longrightarrow e_2[x \mapsto e_1]$

Unfortunately, substitution is rather tricky.

Name capture example

Consider

```
(let x = F in if x then y else S Z)[y ↦ S x]
```

In some outside scope, `x` refers to a natural number.

After naively substituting, we obtain

```
let x = F in if x then S x else S Z
```

which is not even type correct anymore.

Alpha renaming

Fortunately, we can always avoid name capture by consistently renaming:

```
(let x' = F in if x' then y else S Z)[y ↦ S x]
```

This is fine:

```
let x' = F in if x' then S x else S Z
```

Substitution

Interesting are variables and let:

```
x[x ↦ e] = e
y[x ↦ e] = y   -- if y ≠ x

(let x = e₁ in e₂)[x ↦ e] = let x = e₁ in e₂
(let y = e₁ in e₂)[x ↦ e] = let y = e₁[x ↦ e] in e₂[x ↦ e]
                                  -- if y ≠ x and y not free in e
```

Substitution

The rest is rather uninteresting:

```
Z[x ↦ e]                           = Z
(S e')[x ↦ e]                      = S e'[x ↦ e]
T[x ↦ e]                           = T
F[x ↦ e]                           = F
(e₁ == e₂)[x ↦ e]                  = e₁[x ↦ e] == e₂[x ↦ e]
(if e₁ then e₂ else e₃)[x ↦ e] =
  if e₁[x ↦ e] then e₂[x ↦ e] else e₃[x ↦ e]
```

Free variables

$$
\begin{aligned}
\mathsf{free}(\mathsf{Z}) &= \emptyset \\
\mathsf{free}(\mathsf{S}\ e) &= \mathsf{free}(e) \\
\mathsf{free}(\mathsf{F}) &= \emptyset \\
\mathsf{free}(\mathsf{T}) &= \emptyset \\
\mathsf{free}(e_1 == e_2) &= \mathsf{free}(e_1) \cup \mathsf{free}(e_2) \\
\mathsf{free}(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) &= \mathsf{free}(e_1) \cup \mathsf{free}(e_2) \cup \mathsf{free}(e_3) \\
\mathsf{free}(x) &= \{x\} \\
\mathsf{free}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) &= \mathsf{free}(e_1) \cup (\mathsf{free}(e_2) \setminus \{x\})
\end{aligned}
$$

Implementing substitution

Many options, with different advantages and disadvantages.

Often, a form of “de Bruijn” indices is being used to avoid alpha renaming.

```
let x  = Z in let y  = S x  in x  == y
```

Becomes:

```
let ·  = Z in let ·  = S 0ᵥ in 1ᵥ == 0ᵥ
```

Well-Typed

Haskell expressions with variables

```
data Expr =
    Z
  | S     Expr
  | F
  | T
  | Equal Expr Expr
  | If    Expr Expr Expr
  | Var   Int
  | Let   Expr Expr
```

De Bruijn aware traversal

```haskell
dB :: (Int -> Int -> Expr) -> Expr -> Expr
dB var = go 0
  where
    go i (Var j)          = var i j
    go i (Let e₁ e₂)      = Let (go i e₁) (go (i + 1) e₂)
    go _ Z                = Z
    go i (S e)            = S (go i e)
    go _ F                = F
    go _ T                = T
    go i (Equal e₁ e₂)    = Equal (go i e₁) (go i e₂)
    go i (If e₁ e₂ e₃)    = If (go i e₁) (go i e₂) (go i e₃)
```

Substitution

```
subst :: Expr -> Expr -> Expr
subst e₁ e₂ =
  dB var e₂
  where
    var i j =
      case compare i j of
        EQ -> shift i e₁
        LT -> Var (j - 1)
        GT -> Var j
```

Shifting

```
shift :: Int -> Expr -> Expr
shift n e₂ =
  dB var e₂
  where
    var i j
        | j < i = Var j
        | otherwise = Var (j + n)
```

Example

This was our previously prolematic example:

```
(let x = F in if x then y else S Z)[y ↦ S x]
```

Rephrasing:

```
subst (S (Var 42)) (Let F (If (Var 0) (Var 1) (S Z)))
```

Yields:

```
Let F (If (Var 0) (S (Var 43)) (S Z))
```

Evaluation

```
eval :: Expr -> Maybe WHNF
eval (Var _)     = Nothing
eval (Let e₁ e₂) = eval (subst e₁ e₂)
...
```

Well-Typed

Back to type checking

How are environments affected by our choice of de Buijn indexing?

```
data Env =
    Empty
  | Extend Env Ty
```

```
lookup :: Env -> Int -> Maybe Ty
lookup Empty            _ = Nothing
lookup (Extend _   t) 0 = Just t
lookup (Extend env _) i = lookup env (i - 1)
```

Well-Typed

Interesting cases

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

$$\text{Let}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t_2}$$

```
infer :: Env -> Expr -> Maybe Ty
infer env (Var i) =
  lookup env i
infer env (Let e₁ e₂) = do
  t₁ <- infer env e₁
  infer (Extend env t₁) e₂
```

Other cases

$$\text{If}\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : t}$$

```
infer env (If e₁ e₂ e₃) = do
  check env e₁ TBool
  a <- infer env e₂
  check env e₃ a
  pure a
```

Adding functions (Version 3)

Abstraction and application

```
e ::= ...
   |  \ x -> e
   |  e₁ e₂
t ::= ...
   |  t₁ -> t₂
```

$$\text{App}\ \frac{\Gamma \vdash e_1 : t_1 \texttt{ -> } t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

$$\text{Lam}\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \backslash\ x \texttt{ -> } e : t_1 \texttt{ -> } t_2}$$

What is somewhat strange about the abstraction rule?

Type inference

Without knowing the type of `x` in `\ x -> e`, we have to make a guess in the implementation.

This is in essence how type inference systems work:

- introduce a metavariable as a placeholder,
- collect constraints from how the variable of that type (in this case `x` is used),
- apply **unification** to reconcile all the constraints.

Explicit type annotations

```
e ::= ...
    |  \ (x : t) -> e
    |  e₁ e₂
```

$$\text{Lam}\ \frac{\Gamma,\ x : t_1 \vdash e : t_2}{\Gamma \vdash \texttt{\textbackslash}\ (x : t_1) \to e : t_1 \to t_2}$$

Evaluation rules

```
w ::= ...
   |  \ (x : t) -> e
```

s-Beta $\texttt{(\textbackslash\ (x : t) -> } e_1\texttt{)}\ e_2 \longrightarrow \ \textbf{let}\ x = e_2\ \textbf{in}\ e_1$

$$\text{s-App-1}\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

Implementation of functions

```
data Expr =
    ...
  | Lam Ty Expr
  | App Expr Expr
```

```
data Ty =
    ...
  | TFun Ty Ty
```

```
data WHNF =
    ...
  | WLam Ty Expr
```

Evaluation

```
eval :: Expr -> Maybe WHNF
eval (App e₁ e₂) = do
  WLam _ e₁' <- eval e₁
  eval (subst e₂ e₁')
eval (Lam t e)   = pure (WLam t e)
...
```

```
dB :: (Int -> Int -> Expr) -> Expr -> Expr
dB var = go 0
  where
    go i (App e₁ e₂) = App (go i e₁) (go i e₂)
    go i (Lam t e)   = Lam t (go (i + 1) e)
    ...
```

Type checking

```
infer :: Env -> Expr -> Maybe Ty
infer env (App e₁ e₂) = do
  TFun t₁ t₂ <- infer env e₁
  check env e₂ t₁
  pure t₂
infer env (Lam t₁ e) = do
  t₂ <- infer (Extend env t₁) e
  pure (TFun t₁ t₂)
...
```

Simply Typed Lambda Calculus

```
e ::= ...
   |  let x = e₁ in e₂
   |  x
   |  \ x -> e
   |  e₁ e₂
```

```
t ::= ...
   |  t₁ -> t₂
```

The system that has only variables, abstraction and application and function types is called the **simply typed lambda calculus**.

We have now built an extension of the simply typed lambda calculus.

Recursion (Version 4)

Introduction and elimination forms

Data types usually come with two forms of language features:

- constructs that **introduce** an expression of that type,
- constructs that **eliminate** an expression of that type.

Example

Functions:

- introduced via lambda: `\ x -> e` ,
- eliminated via application: $e_1\ e_2$.

Booleans:

- introduced via their constructors `F` and `T` ,
- eliminated via if-then-else: **if** e_1 **then** e_2 **else** e_3 .

Revisiting natural numbers

Natural numbers:

- introduced via their constructors `Z` and `S`,
- eliminated via . . . `==` ?

Revisiting natural numbers

Natural numbers:

- introduced via their constructors `Z` and `S` ,
- eliminated via . . . `==` ?

We do not yet have a **proper** elimination form.

We need a construct that provides at least case distinction, and ideally some form of induction.

Adding recursion

We choose to handle general recursion via its own construct (but that is a big choice):

```
e ::= ...
   |  letrec x : t = e₁ in e₂
```

$$\text{Letrec}\ \frac{\Gamma, x : t_1 \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathbf{letrec}\ x : t_1 = e_1\ \mathbf{in}\ e_2 : t_2}$$

Adding recursion

We choose to handle general recursion via its own construct (but that is a big choice):

```
e ::= ...
   |  letrec x : t = e₁ in e₂
```

$$\text{Letrec}\ \frac{\Gamma, x : t_1 \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{letrec}\ x : t_1 = e_1\ \textbf{in}\ e_2 : t_2}$$

Compare with the rule for non-recursive let:

$$\text{Let}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t_2}$$

Evaluation

s-letrec

$$\textbf{letrec}\ x : t_1 = e_1\ \textbf{in}\ e_2 \longrightarrow e_2[x \mapsto \textbf{letrec}\ x : t_1 = e_1\ \textbf{in}\ e_1]$$

Non-termination

It is not difficult to write a non-terminating program with `letrec` :

```
    letrec x : Nat = x in x
⟶  letrec x : Nat = x in x
⟶  letrec x : Nat = x in x
⟶  ...
```

Productive loops

```
    letrec x : Nat = S x in x
⟶  letrec x : Nat = S x in S x
⟶  S (letrec x : Nat = S x in S x)
```

According to our semantics, this result is a value (in weak head normal form).

Strong normalisation

In fact, without a construct such as `letrec`, the simply typed lambda calculus is **strongly normalising**: it is impossible to write programs that have infinite reduction sequences.

Strong normalisation

In fact, without a construct such as `letrec` , the simply typed lambda calculus is **strongly normalising**: it is impossible to write programs that have infinite reduction sequences.

Strong normalisation can be retained if we add recursion in a more careful way, such as a specific induction principle on natural numbers.

Pattern matching on natural numbers

```
e ::= ...
   |  case e₁ of Z -> e₂; S x -> e₃
```

$$\text{CaseNat}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{Nat} \\ \Gamma \vdash e_2 : t \qquad \Gamma, x : \mathsf{Nat} \vdash e_3 : t\end{array}}{\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ Z \to e_2;\ S\ x \to e_3 : t}$$

Evaluation

$$\texttt{case Z of Z -> } e_2\texttt{; S x -> } e_3 \longrightarrow e_2$$

$$\texttt{case S } e_1 \texttt{ of Z -> } e_2\texttt{; S x -> } e_3 \longrightarrow e_3[x \mapsto e_1]$$

$$\frac{e_1 \longrightarrow e_1'}{\texttt{case } e_1 \texttt{ of Z -> } e_2\texttt{; S x -> } e_3 \longrightarrow \texttt{case } e_1' \texttt{ of Z -> } e_2\texttt{; S x -> } e_3}$$

Example

```
letrec
  add : Nat -> Nat -> Nat =
    \ (m : Nat) -> \ (n : Nat) ->
      case m of
        Z    -> n
        S m’ -> S (add m’ n)
in
  add (S (S Z)) (S Z)
```

Recursion and case analysis in Haskell

```
data Expr =
    ...
  | Letrec  Ty Expr Expr
  | CaseNat Expr Expr Expr
```

Evaluation

```
eval :: Expr -> Maybe WHNF
eval (Letrec t e₁ e₂)   = do
  eval (subst (Letrec t e₁ e₁) e₂)
eval (CaseNat e₁ e₂ e₃) = do
  WNat v₁ <- eval e₁
  case v₁ of
    WZ -> eval e₂
    WS e₁' -> eval (subst e₁' e₃)
...
```

Traversal

```haskell
dB :: (Int -> Int -> Expr) -> Expr -> Expr
dB var = go 0
  where
    go i (Letrec t e1 e2)   =
      Letrec t (go (i + 1) e1) (go (i + 1) e2)
    go i (CaseNat e1 e2 e3) =
      CaseNat (go i e1) (go i e2) (go (i + 1) e3)
    ...
```

Checking

```
infer :: Env -> Expr -> Maybe Ty
infer env (Letrec t1 e1 e2) = do
  check (Extend env t1) e1 t1
  t2 <- infer (Extend env t1) e2
  pure t2
infer env (CaseNat e1 e2 e3) = do
  check env e1 TNat
  t <- infer env e2
  check (Extend env TNat) e3 t
  pure t
...
```

Revisiting the example

```
add :: Expr
add =
  Letrec (TNat `TFun` TNat `TFun` TNat)
    (Lam TNat (Lam TNat
      (CaseNat (Var 1)
        (Var 0)
        (S (Var 3 `App` Var 0 `App` Var 1)))))
    (Var 0 `App` (S (S Z)) `App` S Z)
```

Is inferred to be of type `TNat` .

Evaluates to `S (S (S (Z)))` .

Polymorphism (Version 5)

Status

A lot of concepts / types can be added (or encoded) in STLC, but the type system is actually still quite limited.

Status

A lot of concepts / types can be added (or encoded) in STLC, but the type system is actually still quite limited.

For example, if we wanted to add lists, then we quickly feel the need for **polymorphism**.

Type language

```
t ::= Nat
   |  Bool
   |  t₁ -> t₂
```

Type language

```
t ::= Nat
    |  Bool
    |  t₁ -> t₂
```

Goal:

```
       ...
    |  a
    |  ∀ a . t
    |  [t]
```

Type language

```
t ::= Nat
   |  Bool
   |  t₁ -> t₂
```

Goal:

```
      ...
   |  a
   |  ∀ a . t
   |  [t]
```

Implications:

- We need variables / binders / substitution on types.
- There is the potential for types to be ill-formed.

Expression language

Let us ignore lists for now:

```
e ::= ...
   |  \ @a -> e
   |  e @t
```

Expression language

Let us ignore lists for now:

```
e ::= ...
   |  \ @a -> e
   |  e @t
```

Observations:

- We keep everything explicit, as before, to make inference trivial.
- The expression and type languages are no longer separate.

Examples

Polymorphic identity function:

```
\ @a -> \ (x : a) -> x
```

The S combinator:

```
\ @a -> \ @b -> \ @c ->
\ (f : a -> b -> c) ->
\ (g : a -> b) ->
\ (x : a) ->
(f x) (g x)
```

Revisiting environments

We now need contextual information about two kinds of variables.

It is good to keep these together because types mention type variables:

```
ε, @a, @b, @c, f : a -> b -> c, g : a -> b, x : a
```

```
ε, @a, x : a, @b, y : a -> b
```

We do not have to store further info for types.

Environments with type variables

```
Γ ::= ε
    |  Γ, a : Type
    |  Γ, x : t
```

Well-Typed

Rules for types

The beginnings of a **kind** system:

TNat $\Gamma \vdash \mathtt{Nat} : \mathtt{Type}$ TBool $\Gamma \vdash \mathtt{Bool} : \mathtt{Type}$

$$\text{TFun}\ \frac{\Gamma \vdash t_1 : \mathtt{Type} \qquad \Gamma \vdash t_2 : \mathtt{Type}}{\Gamma \vdash t_1 \;\texttt{->}\; t_2 : \mathtt{Type}}$$

$$\text{TVar}\ \frac{a : \mathtt{Type} \in \Gamma}{\Gamma \vdash a : \mathtt{Type}} \qquad \text{TForall}\ \frac{\Gamma, a : \mathtt{Type} \vdash t : \mathtt{Type}}{\Gamma \vdash \forall\, a\,.\, t : \mathtt{Type}}$$

$$\text{TList}\ \frac{\Gamma \vdash t : \mathtt{Type}}{\Gamma \vdash \texttt{[}t\texttt{]} : \mathtt{Type}}$$

Type rules

$$\text{TLam}\ \frac{\Gamma,\ a : \mathrm{Type} \vdash e : t}{\Gamma \vdash \backslash\ @a \texttt{ -> } e : \forall\ a\ .\ t}$$

$$\text{TApp}\ \frac{\Gamma \vdash e : \forall\ a\ .\ t \qquad \Gamma \vdash t' : \mathrm{Type}}{\Gamma \vdash e\ @t' : t[a \mapsto t']}$$

We need substitution on types.

Well-Typed

Revised type rules

Rules where types occur in the syntax now need to check these:

$$\text{Lam}\ \frac{\Gamma,\ x : t_1 \vdash e : t_2 \qquad \Gamma \vdash t_1 : \mathsf{Type}}{\Gamma \vdash \backslash\,(x : t_1) \to e : t_1 \to t_2}$$

Similar for letrec.

Evaluation rules

```
w ::= \ @a -> e
```

$$\texttt{(\textbackslash{} @a e) @t} \longrightarrow \texttt{e[a} \mapsto \texttt{t]}$$

$$\frac{\texttt{e} \longrightarrow \texttt{e'}}{\texttt{e @t} \longrightarrow \texttt{e' @t}}$$

We also need to substitute types in expressions (which in turn can contain types again).

Lists

This is now all more of the same …

```
e ::= Nil t
   |  (Cons e₁ e₂)
   |  case e₁ of Nil -> e₂; Cons x₁ x₂ -> e₃
```

Type rules

$$\textsf{Nil}\ \frac{\Gamma \vdash t : \mathsf{Type}}{\Gamma \vdash \mathtt{Nil}\ t : [t]}$$

$$\textsf{Cons}\ \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : [t]}{\Gamma \vdash \mathtt{Cons}\ e_1\ e_2 : [t]}$$

$$\textsf{CaseList}\ \frac{\Gamma \vdash e_2 : t' \qquad \begin{array}{c}\Gamma \vdash e_1 : [t] \\ \Gamma, x_1 : t, x_2 : [t] \vdash e_3 : t'\end{array}}{\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathtt{Nil}\ \texttt{->}\ e_2;\ \mathtt{Cons}\ x_1\ x_2\ \texttt{->}\ e_3 : t'}$$

Well-Typed

Evaluation rules

$$\texttt{case Nil t of Nil -> } e_2;\ \texttt{Cons } x_1\ x_2 \texttt{ -> } e_3 \longrightarrow e_2$$

$$\frac{e_3' = e_3[x_2 \mapsto e_2'][x_1 \mapsto e_1']}{\texttt{case Cons } e_1'\ e_2' \texttt{ of Nil -> } e_2;\ \texttt{Cons } x_1\ x_2 \texttt{ -> } e_3 \longrightarrow e_3'}$$

$$\frac{e_1 \longrightarrow e_1' \qquad e' = \texttt{case } e_1' \texttt{ of Nil -> } e_2;\ \texttt{Cons } x_1\ x_2 \texttt{ -> } e_3}{\texttt{case } e_1 \texttt{ of Nil -> } e_2;\ \texttt{Cons } x_1\ x_2 \texttt{ -> } e_3 \longrightarrow e'}$$

Implementation

See code . . .

Conclusion

What is next?

A natural extension to System F is System Fω, which basically turns the type language into something akin to the Simply Typed Lambda Calculus, by adding **function kinds**.

Other interesting related topics

- Proper type inference
- Dependent types
- Type classes; adding translation aspects to a type system
- More efficient implementations (abstract machines; code generation)
- Other / better ways to handle variables
- ...