An Implementation of Type checking for a Dependently Typed Lambda Calculus

Based on:

A tutorial implementation of a dependently typed lambda calculus

A. Löh, C. McBride, W. Swierstra

What even are Dependent Types?

- The normal Function type $\tau \to \tau'$ is extended to $\forall x : \tau . \tau'$
- Also $(x : \tau) \to \tau'$ or $\Pi_{x:\tau}\tau'(x)$
- Return type τ' can depend *value* of argument $x : \tau$
- Like polymorphism, but for all values, not just types

```haskell
-- The `cons` function for Lists and Vectors

cons_monomorphic :: Int → [Int] → [Int]

cons_polymorphic :: forall a. a → [a] → [a]

cons_dependent :: forall (a :: *) (n :: Int). Vec a n → Vec a (n + 1)
--^ This sadly is not legal Haskell
```

What even are Dependent Types *for*?

- General Functional Programming
  - **Idris**
  - Stronger compile time invariants
- Proof Assistant
  - **Agda**, **Coq**, **Lean**
  - Automatically check the correctness of proofs
  - $\longrightarrow$ *Curry–Howard Correspondence*

Proving things with Dependent Types

```
-- Associativity of addition on natural numbers in Agda

data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

data _≡_ : Nat → Nat → Set where
  refl : {x : Nat} → x ≡ x

_+_ : Nat → Nat → Nat
zero    + y = y
(suc x) + y = suc (x + y)

assoc : (x : Nat) → (y : Nat) → (z : Nat) → ((x + y) + z) ≡ (x + (y + z))
assoc x y z = ?
```

Programming with Dependent Types

```
-- Known-length vectors and the functions `append` and `head` on them

data Vec : Set → Nat → Set where
    nil  : {A : Set} → Vec A zero
    _::_ : {A : Set} → {n : Nat} → (a : A) → Vec A n → Vec A (suc n)

append : {A : Set} → {n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
append nil      v' = v'
append (x :: v) v' = x :: (append v v')

head : {A : Set} → {n : Nat} → Vec A (suc n) → A
head (x :: v) = x
```

Inputs and Outputs in Type Judgements

$$\frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \tau'}$$

- How do we translate this into code? What is *input*? What is *output*?
- $\to$ **Type Checking** vs **Type Inferece** (vs **Program Synthesis**)
- $\Rightarrow$ We differentiate between:
  - $\Gamma \vdash e :_{\downarrow} \tau$ ("Check that e has given type τ, in context Γ")
  - $\Gamma \vdash e :_{\uparrow} \tau$ ("Infer for e what type τ it has, in context Γ")

Bindings...

$$(\lambda x.\, \lambda y.\, x)(\lambda y.\, y)$$
$$\rightsquigarrow \lambda y.\, \lambda y.\, y \qquad\qquad (. \_ . ?)$$

- There is no silver bullet solution
- We allow for two styles of bindings (→ *locally nameless*)
  - Local: *de Bruijn indices*
  - Global: String names
- For example:
  - $id = \lambda x.\, x = \lambda\, 0$
  - $const = \lambda x.\, \lambda y.\, x = \lambda\, \lambda\, 1$

Abstract syntax STLC

$$
\begin{aligned}
e ::= \quad & e : \tau \\
\mid \quad & x \\
\mid \quad & e\ e' \\
\mid \quad & \lambda x.\, e
\end{aligned}
$$

$$
\begin{aligned}
\tau ::= \quad & \alpha \\
\mid \quad & \tau \to \tau'
\end{aligned}
$$

```
data TermInfer
  = Ann TermCheck Type
  | Bound Int
  | Free Name
  | TermInfer :@: TermCheck

data TermCheck
  = Inf TermInfer
  | Lam TermCheck

data Type
  = TFree Name
  | Fun Type Type
```

Abstract syntax DTLC

$$
\begin{aligned}
e, \rho, \kappa ::= \quad & e : \rho \\
\mid \quad & * \\
\mid \quad & \forall x : \rho.\, \rho' \\
\mid \quad & x \\
\mid \quad & e\ e' \\
\mid \quad & \lambda x.\, e
\end{aligned}
$$

```haskell
data TermInfer
  = Ann TermCheck TermCheck
  | Star
  | Pi TermCheck TermCheck
  | Bound Int
  | Free Name
  | TermInfer :@: TermCheck

data TermCheck
  = Inf TermInfer
  | Lam TermCheck
```

Type Checking of Inferrable Term

$$\frac{\Gamma \vdash e :_{\uparrow} \tau}{\Gamma \vdash e :_{\downarrow} \tau}$$

```
typeCheck i g (Inf e) t = do
  t' ← typeInfer i g e
  if t == t'
    then return ()
    else
      failure ":("
```

Type Inference of Free Variables

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :_{\uparrow} \tau}$$

```
typeInfer i g (Free x) =
  case lookup x g of
    Just t → return t
    Nothing → failure ":("
```

Type Checking of Abstraction ($\lambda x.\, e$)

$$\frac{\Gamma, x : \tau \vdash e :_{\downarrow} \tau'}{\Gamma \vdash \lambda x.\, e :_{\downarrow} \tau \to \tau'}$$

```
typeCheck i g (Lam e) (Fun t t') =
  typeCheck (i + 1)
    ((Local i, HasType t) : g)
    (substCheck 0 (Free (Local i)) e)
    t'
```

$$\frac{\Gamma, x : \tau \vdash e :_{\downarrow} \tau'}{\Gamma \vdash \lambda x.\, e :_{\downarrow} \forall x : \tau.\, \tau'}$$

```
typeCheck i g (Lam e) (VPi t t') =
  typeCheck (i + 1)
    ((Local i, t) : g)
    (substCheck 0 (Free (Local i)) e)
    (t' (vfree (Local i)))
```

Type Inference of Application ($e\ e'$)

$$\frac{\Gamma \vdash e :_{\uparrow} \tau \to \tau' \qquad \Gamma \vdash e' :_{\downarrow} \tau}{\Gamma \vdash e\ e' :_{\uparrow} \tau'}$$

```
typeInfer i g (e :@: e') = do
  s ← typeInfer i g e
  case s of
    Fun t t' → do
      typeCheck i g e' t
      return t'
    _ → failure ":("
```

$$\frac{\Gamma \vdash e :_{\uparrow} \forall x : \tau . \tau' \qquad \Gamma \vdash e' :_{\downarrow} \tau}{\Gamma \vdash e\ e' :_{\uparrow} \tau[x \mapsto e']}$$

```
typeInfer i g (e :@: e') = do
  s ← typeInfer i g e
  case s of
    VPi t t' → do
      typeCheck i g e' t
      return
        (t' (evalCheck [] e'))
    _ → failure ":("
```

Type Inference of Annotation ($e : \rho$)

$$\frac{\Gamma \vdash \tau : * \qquad \Gamma \vdash e :_{\downarrow} \tau}{\Gamma \vdash (\, e : \tau \,) :_{\uparrow} \tau}$$

```
typeInfer i g (Ann e t) = do
  kindCheck g t Star
  typeCheck i g e t
  return t
```

$$\frac{\Gamma \vdash \rho :_{\downarrow} * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_{\downarrow} \tau}{\Gamma \vdash (\, e : \rho \,) :_{\uparrow} \tau}$$

```
typeInfer i g (Ann e r) = do
  typeCheck i g r VStar
  let t = evalCheck [] r
  typeCheck i g e t
  return t
```

Kinding of Types ($\tau \to \tau'$ and $\forall x : \rho.\, \rho'$)

$$\frac{\Gamma \vdash \tau : * \qquad \Gamma \vdash \tau' : *}{\Gamma \vdash \tau \to \tau' : *}$$

```
kindCheck g (Fun k k') Star = do
  kindCheck g k Star
  kindCheck g k' Star
```

$$\frac{\Gamma \vdash \rho :_{\downarrow} * \qquad \rho \Downarrow \tau \qquad \Gamma, x : \tau \vdash \rho' :_{\downarrow} *}{\Gamma \vdash \forall x : \rho.\, \rho' :_{\uparrow} *}$$

```
typeInfer i g (Pi r r') =
  do
    typeCheck i g r VStar
    let t = evalCheck [] r
    typeCheck (i + 1)
      ((Local i, t) : g)
      (substCheck 0
        (Free (Local i)) r')
      VStar
    return VStar
```

The Type of the Type of Types

$$\frac{}{\Gamma \vdash * :_{\uparrow} *}$$

```
typeInfer i g Star =
  return VStar
```

- This is *unsound* (→ *Girard's paradox*)
- ⇒ Idea: Introduce a hierarchy of *sorts*
  - $* : *_1$
  - $*_1 : *_2$
  - $*_2 : *_3$
  - …

Conclusion

- Dependent types aren't scary
- Implementing type inference & checking neither
-

Sources & co

Slides at: https://github.com/Garbaz/seminar-dependent-types

[1] Löh, Andres, Conor McBride, Wouter Swierstra. *"A tutorial implementation of a dependently typed lambda calculus."* Fundamenta informaticae 102.2 (2010): 177-207.
[2] Jana Dunfield, Neel Krishnaswami. *"Bidirectional typing"* ACM Computing Surveys (CSUR) 54.5 (2021): 1-38.