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 $(x :: \tau) \to \tau'$
- Also commonly written as $\forall x :: \tau.\, \tau'$ or $\Pi_{x::\tau}\tau'(x)$
- The return type *(range)* can now depend on the argument type *(domain)*
- Like polymorphism, but for all values, not just types

```
-- The `cons` function for lists and vectors

cons_monomorphic :: Int → List Int → List Int

cons_polymorphic :: (a :: *) → List a → List a

cons_dependent :: (n :: Nat) → (a :: *) → Vec a n → Vec a (1 + n)
```

Dependent Types in Practice

- General Functional Programming
  - **Idris**
  - Stronger compile time invariants
- Proof Assistant
  - **Agda, Coq**
  - Automatically check the correctness of proofs

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 0
    _::_ : {a : Set} → {n : Nat} → a → Vec a n → Vec a (1 + 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} → {1 ≤ n} → Vec a n → a
head (x :: v) = x
```

Proving things with Dependent Types

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

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

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

data _≡_ (x : Nat) → Set where
  refl : x ≡ x

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

Inputs and Outputs in Typing Judgements

...

=> Differentiate between inferrable Terms and checkable Terms

...

Abstract syntax STLC

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

```haskell
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

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

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

data TermCheck
  = Inf TermInfer
  | Lam TermCheck
```

((Something simple here as intro))

...

Interlude: Bound Variables 😬

- There is no silver bullet solution
- We use a combintation of two styles of bindings (→ *locally nameless*)
  - Local: *de Bruijn indices*
  - Global: *String names*
- E.g.: $const = \lambda \to \lambda \to 1$

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

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

```
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 ":("
```

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

```
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 ($e :: \rho$)

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

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

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

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

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

…

Issues & Extensions

...

Conclusion

- Dependent types aren't as scary as they seem
- ...

Sources & co

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

[1] Löh, Andres, Conor McBride, and Wouter Swierstra. *"A tutorial implementation of a dependently typed lambda calculus."* Fundamenta informaticae 102.2 (2010): 177-207.