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 $\tau \rightarrow \tau'$ is extended to $\forall x : \tau . \tau'$
- Also commonly written as $(x : \tau) \rightarrow \tau'$ or $\Pi_{x:\tau}\tau'(x)$
- The return type τ' can now depend on the argument *value* $x : \tau$
- 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
  - $\longrightarrow$ *Curry–Howard correspondence*

Programming with Dependent Types

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

data Vec (A : Set) : ℕ → Set where
    nil  : Vec A 0
    _::_ : {n : ℕ} → (a : A) → Vec A n → Vec A (1 + n)

append : {A : Set} → {n m : ℕ} → 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 : ℕ} → {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 Type Judgements

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

- ...

Inputs and Outputs in Type Judgements

- In some Type Judgement $\Gamma \vdash e : \tau$, what is given, and what is inferred?
- Γ, e and τ are given ⇒ *Type Checking*
- Γ and e are given ⇒ *Type Inference*
- (Γ and τ are given ⇒ *Program Sythesis*)
- ⇒ 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")

Abstract syntax STLC

$$
\begin{aligned}
e ::= \quad & e : \tau \\
\mid \quad & x \\
\mid \quad & e\ e' \\
\mid \quad & \lambda x.\, e \\[1em]
\tau ::= \quad & \alpha \\
\mid \quad & \tau \to \tau'
\end{aligned}
$$

```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

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

```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 Abstraction ($\lambda x.\, e$)

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

```
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)))
```

Interlude: Bound Variables 😬

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

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 ($e : \rho$)

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

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

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

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

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

...

Issues & Extensions

- What is the type of the type of types? I.e. For what τ is $* : \tau$?
- For simplicity we assumed $* : *$, but this is *unsound*
- ...

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, 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.