Normalization or bust

Stuart Gale & Conor McBride

October 13, 2017

Motivation

- Proof of concept for dependently typed/generic system
- No fancy techniques we can't scale up eg type safe terms
- 'Basic' training for me

Simple types synthesized: Syntax, Typing and Reduction

Syntax

$$
\begin{array}{lrcl}
Types & \sigma, \tau & ::= & \iota \mid \sigma \Rightarrow \tau \\
Contexts & \Gamma & ::= & \cdot \mid \Gamma, x : \sigma
\end{array}
$$

$$
\begin{array}{lrcl}
Terms & f, s, t & ::= & x \mid f\ s \mid \lambda x : \sigma.\ t[x]
\end{array}
$$

Typing (1/3)

$$\boxed{\Gamma \vdash t : \tau}$$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash f : \sigma \Rightarrow \tau \quad \Gamma \vdash s : \sigma' \quad \sigma = \sigma'}{\Gamma \vdash f\ s : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t[x] : \tau}{\Gamma \vdash \lambda x : \sigma.\ t[x] : \sigma \Rightarrow \tau}$$

Reduction

$$(\lambda x : \sigma.\ t[x])\ s \rightsquigarrow_\beta t[s]$$

Typing (2/3)

$$\boxed{\Gamma \vdash t : \tau}$$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash f : \sigma \Rightarrow \tau \quad \Gamma \vdash s : \sigma' \quad \sigma = \sigma'}{\Gamma \vdash f\ s : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t[x] : \tau}{\Gamma \vdash \lambda x : \sigma.\ t[x] : \sigma \Rightarrow \tau}$$

Reduction

$$(\lambda x : \sigma.\ t[x])\ s \rightsquigarrow_\beta t[s]$$

Typing (3/3)

$$\boxed{\Gamma \vdash t : \tau}$$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash f : \sigma \Rightarrow \tau \quad \Gamma \vdash s : \sigma' \quad \sigma = \sigma'}{\Gamma \vdash f\ s : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t[x] : \tau}{\Gamma \vdash \lambda x : \sigma.\ t[x] : \sigma \Rightarrow \tau}$$

Reduction

$$(\lambda x : \sigma.\ t[x])\ s \rightsquigarrow_\beta t[s]$$

Bidirectional changes

Bidirectional syntax changes

$$
\begin{array}{lrcl}
\mathit{Types} & \sigma, \tau & ::= & \iota \mid \sigma \Rightarrow \tau \\
\mathit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \sigma \\
 & & & \\
\mathit{Synthesizable} & e, f & ::= & x \mid f\ s \\
\mathit{Checkable} & s, t & ::= & \lambda x.\ t[x] \mid \underline{e}
\end{array}
$$

Bidirectional syntax changes

$$
\begin{array}{lrcl}
\textit{Types} & \sigma, \tau & ::= & \iota \mid \sigma \Rightarrow \tau \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \sigma \\
 & & & \\
\textit{Synthesizable} & e, f & ::= & x \mid f\ s \mid t : \tau \\
\textit{Checkable} & s, t & ::= & \lambda x.\ t[x] \mid \underline{e}
\end{array}
$$

Syntax in Agda

```
data Ty : Set where
  base : Ty
  _=>_ : Ty -> Ty -> Ty

data Dir : Set where chk syn : Dir

data Tm (n : Nat) : Dir -> Set where
  var  : (i : Fin n)                    -> Tm n syn
  _$_  : (f : Tm n syn)(s : Tm n chk)   -> Tm n syn
  _::_ : (t : Tm n chk)(T : Ty)         -> Tm n syn

  [_]  : (e : Tm n syn)                 -> Tm n chk
  lam  : (t : Tm (ns n) chk)            -> Tm n chk
```

Bidirectional typing

$$\boxed{\Gamma \vdash e \in \ \sigma} \qquad \boxed{\Gamma \vdash \tau \ni \ t}$$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x \in \ \sigma} \qquad \frac{\Gamma \vdash \sigma \ni \ s}{\Gamma \vdash s : \sigma \in \ \sigma}$$

$$\frac{\Gamma \vdash f \in \ \sigma \Rightarrow \tau \quad \Gamma \vdash \sigma \ni \ s}{\Gamma \vdash f \ s \in \ \tau}$$

$$\frac{\Gamma, x : \sigma \vdash \tau \ni \ t[x]}{\Gamma \vdash \sigma \Rightarrow \tau \ni \ \lambda x.\ t[x]}$$

$$\frac{\Gamma \vdash e \in \ \sigma \quad \sigma = \tau}{\Gamma \vdash \tau \ni \ \underline{e}}$$

Bidirectional reduction

$$(\lambda x.\ t[x] : \sigma \Rightarrow \tau)\ s \rightsquigarrow_\beta t[s : \sigma] : \tau \qquad \underline{t : \tau} \rightsquigarrow_\upsilon t$$

Example: Omega (1/5)

$$(\lambda x.\ t[x] : \sigma \Rightarrow \tau)\ s \rightsquigarrow_\beta t[s : \sigma] : \tau \qquad \underline{t : \tau} \rightsquigarrow_\upsilon t$$

$$\begin{aligned} E_0 &= \iota \\ E_{n+1} &= E_n \Rightarrow E_n \\ \omega &= \lambda x.\ \underline{x\ \underline{x}} \end{aligned} \qquad \Omega = \underline{(\omega : E_{n+1})\ \omega}$$

Example: Omega (2/5)

$$(\lambda x.\ t[x] : \sigma \Rightarrow \tau)\ s \rightsquigarrow_\beta t[s : \sigma] : \tau \qquad \underline{t : \tau} \rightsquigarrow_\upsilon t$$

$$\begin{aligned} E_0 &= \iota \\ E_{n+1} &= E_n \Rightarrow E_n \\ \omega &= \lambda x.\ \underline{x\ \underline{x}} \end{aligned} \qquad \begin{aligned} \Omega &= \underline{(\omega : E_{n+1})\ \omega} \\ &= \underline{(\lambda x.\ \underline{x\ \underline{x}} : E_n \Rightarrow E_n)\ \omega} \end{aligned}$$

Example: Omega (3/5)

$$(\lambda x.\ t[x] : \sigma \Rightarrow \tau)\ s \rightsquigarrow_\beta t[s : \sigma] : \tau \qquad \underline{t : \tau} \rightsquigarrow_\upsilon t$$

$$\begin{aligned} E_0 &= \iota \\ E_{n+1} &= E_n \Rightarrow E_n \\ \omega &= \lambda x.\ \underline{x\ \underline{x}} \end{aligned}$$

$$\begin{aligned} \Omega &= \underline{(\omega : E_{n+1})\ \omega} \\ &= \underline{(\lambda x.\ \underline{x\ \underline{x}} : E_n \Rightarrow E_n)\ \omega} \\ &\rightsquigarrow_\beta \underline{\underline{(\omega : E_n\ )(\underline{\omega : E_n}) : E_n}} \end{aligned}$$

Example: Omega (4/5)

$$(\lambda x.\ t[x] : \sigma \Rightarrow \tau)\ s \rightsquigarrow_\beta t[s : \sigma] : \tau \qquad \underline{t : \tau} \rightsquigarrow_\upsilon t$$

$$\begin{aligned} E_0 &= \iota \\ E_{n+1} &= E_n \Rightarrow E_n \\ \omega &= \lambda x.\ \underline{x\ \underline{x}} \end{aligned}$$

$$\begin{aligned} \Omega &= \underline{(\omega : E_{n+1})\ \omega} \\ &= \underline{(\lambda x.\ \underline{x\ \underline{x}} : E_n \Rightarrow E_n)\ \omega} \\ &\rightsquigarrow_\beta \underline{\underline{(\omega : E_n\ )\underline{(\omega : E_n)}} : E_n} \\ &\rightsquigarrow_\upsilon \underline{(\omega : E_n\ )\underline{(\omega : E_n)}} \end{aligned}$$

Example: Omega (5/5)

$$(\lambda x.\ t[x] : \sigma \Rightarrow \tau)\ s \rightsquigarrow_\beta t[s : \sigma] : \tau \qquad \underline{t : \tau} \rightsquigarrow_\upsilon t$$

$$\begin{aligned} E_0 &= \iota \\ E_{n+1} &= E_n \Rightarrow E_n \\ \omega &= \lambda x.\ \underline{x\ \underline{x}} \end{aligned}$$

$$\begin{aligned} \Omega &= \underline{(\omega : E_{n+1})\ \omega} \\ &= \underline{(\lambda x.\ \underline{x\ \underline{x}} : E_n \Rightarrow E_n)\ \omega} \\ &\rightsquigarrow_\beta \underline{\underline{(\omega : E_n)\ \underline{(\omega : E_n)}} : E_n} \\ &\rightsquigarrow_\upsilon \underline{\underline{(\omega : E_n)\ \underline{(\omega : E_n)}}} \\ &\rightsquigarrow_\upsilon \underline{(\omega : E_n)\ \omega} \end{aligned}$$

Normalization or bust

$\beta\upsilon$-normal forms (excluding obviously bad)

```
data Nm (n : Nat) : Dir -> Set where
  var       : (i : Fin n)                                -> Nm n syn
  _$_       : (f : Nm n syn)(s : Nm n chk)               -> Nm n syn
  [_]::_:$_ : (f : Nm n syn)(T : Ty)(s : Nm n chk)       -> Nm n syn

  lam       : (t : Nm (ns n) chk)                        -> Nm n chk
  [_]       : (e : Nm n syn)                             -> Nm n chk
```

Values

```
Val : Ty -> Nat -> Set
Go : Ty -> Nat -> Set

Val T n = (Nm n syn) + Go T n
Go base      n = Zero
Go (S => T)  n = (m : Nat) ->
                 n <= m ->
                 Val S m ->
                 Maybe (Val T m)



Cell : Nat -> Set
Cell n = Sg Ty \ T -> Val T n

Env : Nat -> Nat -> Set
Env n m = Vec (Cell m) n
```

```
norm : forall {n} ->
       Cx n -> Tm n syn -> Maybe (Ty * Nm n chk)
norm G e =
  evalSyn (idEnv G) e  >>= \ { (T , v) ->
  reifyVal v           >>= \ t ->
  yes (T , t) }
```

```
evalSyn : forall {n m} ->
          (Ga : Env n m)(s : Tm n syn) ->
          Maybe (Cell m)

evalSyn Ga (var i)  = yes (Ga ! i)
evalSyn Ga (f $ s)  =
  evalSyn Ga f    >>= \
  { (base   , _)  -> no
  ; (S => T , f') -> chkEval Ga S s  >>= \ s' ->
                     applyVal f' s'  >>= \ v ->
                     yes (T , v)
  }
evalSyn Ga (t :: T) =
  chkEval Ga T t >>= \ t' ->
  yes (T , t')
```

```
chkEval : forall {n m : Nat} ->
          (Ga : Env n m)(T : Ty)(c : Tm n chk) ->
          Maybe (Val T m)

chkEval Ga T          [ e ]   =
  evalSyn Ga e >>= \ { (S , v) ->
  S =Ty? T     >>= \ { refl -> yes v } }
chkEval Ga base       (lam t) = no
chkEval Ga (S => T)   (lam t) = yes (inr \ m th s ->
  chkEval (actEnv th Ga -, (S , s)) T t)
```

```
applyVal : forall {n S T} ->
           Val (S => T) n -> Val S n -> Maybe (Val T n)
applyVal (inl f) s = reifyVal s >>= \ s' ->
                     yes (inl (f $ s'))
applyVal (inr g) s = g _ iT s


reifyVal : forall {T n} -> Val T n -> Maybe (Nm n chk)
reifyGo  : forall {n}(T : Ty) -> Go T n -> Maybe (Nm n chk)

reifyVal (inl e) = yes [ e ]
reifyVal (inr g) = reifyGo _ g

reifyGo base     ()
reifyGo (S => T) g =
  g _ (o' iT) (inl (var fz))  >>= \ v ->
  reifyVal v                  >>= \ t ->
  yes (lam t)
```

Reviewing the big picture

- Don't need to statically type check functions to run them
- Well typed programs don't go bust!
- Enough information attached to the terms, via radicals, that dynamic checking is sufficient
- Values say how much computation a stated type will allow, and values are defined by structural recursion on types

Future work

- Done: strong confluence, type preservation, progress
- In the middle of: weak normalization
- To do: strong normalization
- Informative assignment of blame for incorrect terms
- Subtyping
- Extending to fancier (predicative) systems

Questions?