How to reify fresh type variables?

Oleg Kiselyov       Chung-chieh Shan

15 September 2011
Waiter, there’s a term in my type!

Rank-2 polymorphism (ab)used, for safety:

- mutable state
- array index bounds
- environment classifiers
- staged lexical scope
- resource control
- automatic differentiation

\[
\begin{align*}
\alpha : \star \\
\vdots \\
E : \tau \\
\Lambda \alpha. E : \forall \alpha. \tau \\
\end{align*}
\]

\(\alpha\) is fresh, but the program doesn’t know it.

Type-level gensyms for expressivity?
Waiter, there’s a term in my type!

- staged lexical scope
- resource control

$\forall$, see you later.
Waiter, there’s a term in my type!

- staged lexical scope
- resource control

∀, see you later.
Effects beyond generated binders

Don’t want

- syntax errors
- type errors
- unexpectedly unbound variables
- unexpectedly bound variables
Effects beyond generated binders

Don’t want

- syntax errors
- type errors
- unexpectedly unbound variables
- unexpectedly bound variables

R. Clint Whaley, ATLAS documentation:

You may have a naturally strong and negative reaction to these crude mechanisms, tempting you to send messages decrying my lack of humanity, decency, and legal parentage… The proper bitch format involves

First thanking me for spending time in hell getting things to their present crude state

Then, supplying your constructive ideas
Higher-order abstract syntax

- lam (\x -> x) \rightarrow
  Lam "x1" (Var "x1")

- lam (\x -> let body = x in lam (\x -> body)) \rightarrow
  Lam "x2" (let body = Var "x2" in lam (\x -> body)) \rightarrow
  Lam "x2" (lam (\x -> Var "x2")) \rightarrow
  Lam "x2" (Lam "x3" (Var "x2"))
Higher-order abstract syntax

▶ \( \text{lam} (\,x \to x) \mapsto\) \\
\hspace{1cm} \text{Lam} "x1" (\text{Var} "x1")

▶ \( \text{lam} (\,x \to \text{let body} = x \text{ in lam} (\,x \to \text{body})) \mapsto\) \\
\hspace{1cm} \text{Lam} "x2" (\text{let body} = \text{Var} "x2" \text{ in lam} (\,x \to \text{body})) \mapsto \\
\hspace{1cm} \text{Lam} "x2" (\text{lam} (\,x \to \text{Var} "x2")) \mapsto \\
\hspace{1cm} \text{Lam} "x2" (\text{Lam} "x3" (\text{Var} "x2"))

Effects (error, state, let-insertion, etc.) beyond binders are hard.

▶ \( \text{lam} (\,x \to \text{throw} "\text{hello}\") \mapsto ???\)

▶ \( \text{lam} (\,x \to \text{throw} \,x) \mapsto ???\)

*It seems rather difficult, if not impossible, to manipulate open code in a satisfactory manner when higher-order code representation is chosen.* (Chen & Xi, JFP 2005)

We need name generation, but dissociated from binding.
Gensym

- `let x = gensym() in Lam x (Var x) ⇝ Lam "x1" (Var "x1")`

- `let x = gensym() in Lam x
  (let body = Var x in
   let x = gensym() in Lam x body) ⇝ Lam "x2" (Lam "x3" (Var "x2"))`

- `let x = gensym() in cogen (fun body -> Lam x body) ⇝`
Gensym

- let $x = \text{gensym}()$ in $\text{Lam} \ x \ (\text{Var} \ x) \Rightarrow \text{Lam} \ "x1" \ (\text{Var} \ "x1")$

- let $x = \text{gensym}()$ in $\text{Lam} \ x$
  
  (let body = Var $x$ in
  
  let $x = \text{gensym}()$ in $\text{Lam} \ x$ body) \Rightarrow

  $\text{Lam} \ "x2" \ (\text{Lam} \ "x3" \ (\text{Var} \ "x2"))$

- let $x = \text{gensym}()$ in $\text{cogen}$ (fun body -> Lam $x$ body) \Rightarrow

  Ruling out scope extrusion is hard.

- let $x = \text{gensym}()$ in Lam $x$ (throw "hello") \Rightarrow

- let $x = \text{gensym}()$ in Lam $x$ (throw (Var $x$)) \Rightarrow
So, de Bruijn

- Lam Zero

- Lam (let body = Zero in Lam (Succ body)) \implies Lam (Lam (Succ Zero))

- let x = Zero in cogen (fun body \to Lam body) \implies
So, de Bruijn

- Lam Zero

- Lam (let body = Zero in Lam (Succ body)) \leadsto Lam (Lam (Succ Zero))

- let x = Zero in cogen (fun body -> Lam body) \leadsto

Mourn the loss of HOAS beauty.

Meta-types should reflect object type judgments (Nanevski, Pfenning & Pientka, TOCL 2008).

\[
\begin{align*}
\text{Zero: } & (\Gamma, \text{Int} \vdash \text{Int}) \\
\text{Succ Zero: } & (\Gamma, \text{Int}, \text{Bool} \vdash \text{Int}) \\
\text{Lam (Succ Zero): } & (\Gamma, \text{Int} \vdash \text{Bool} \to \text{Int}) \\
\text{Lam (Lam (Succ Zero)): } & (\Gamma \vdash \text{Int} \to \text{Bool} \to \text{Int})
\end{align*}
\]
Type safety

Open code and closed code have distinct types:

\[
\begin{align*}
&\text{catch (throw (Lam Zero))}: (\text{Int} \rightarrow \text{Int}) \\
&\text{run (catch (throw (Lam Zero))))}: \text{Int} \rightarrow \text{Int} \\
&\text{catch (Lam (throw "hello"))}: \text{String} \\
&\text{catch (Lam (throw Zero))}: (\Gamma, \text{Int} \vdash \text{Int}) \\
&\text{catch (Lam (throw Zero))}: (\text{Int} \vdash \text{Int}) \\
&\text{Lam (catch (Lam (throw Zero))))}: (\text{Int} \rightarrow \text{Int}) \\
&\text{run (Lam (catch (Lam (throw Zero)))))): \text{Int} \rightarrow \text{Int} \\
\end{align*}
\]

(Kim, Yi & Calcagno, POPL 2006, §6.4)

Where did lexical scope go?
Unexpectedly bound variables

\[
\text{uneasy } f = \text{Lam (Lam (f Zero))} \quad \text{(Chen & Xi, JFP 2005)}
\]

\[
\begin{array}{l}
\text{uneasy id} \mapsto \text{Lam (Lam Zero)} \\
\text{uneasy Succ} \mapsto \text{Lam (Lam (Succ Zero))} \\
\text{uneasy (fun body } \to \text{ Lam (Succ body))} \mapsto \\
\text{Lam (Lam (Lam (Succ Zero)))}
\end{array}
\]

\textit{In light of these examples, we claim that, perhaps contrary to popular belief, well-scopedness of de Bruijn indices is not good enough: it does not guarantee that indices are correctly adjusted where needed.}

\text{(Pouillard & Pottier, ICFP 2010)}
Unexpectedly bound variables

uneasy $f = \text{Lam (Lam (f \text{ Zero}))}$  
  (Chen & Xi, JFP 2005)

▶ uneasy $\text{id} \leadsto \text{Lam (Lam Zero)}$

▶ uneasy $\text{Succ} \leadsto \text{Lam (Lam (Succ Zero))}$

▶ uneasy $(\text{fun body -> Lam (Succ body)}) \leadsto$
  \text{Lam (Lam (Lam (Succ Zero)))}$

\textit{In light of these examples, we claim that, perhaps contrary to popular belief, well-scopedness of de Bruijn indices is not good enough: it does not guarantee that indices are correctly adjusted where needed.}

  (Pouillard & Pottier, ICFP 2010)
FREE BEER
TOMORROW
Safety in numbers

- let x = gensym() in Lam x (Zero x) ⇝ Lam 1 (Zero 1)

- let x = gensym() in Lam x
  (let body = Zero x in
   let x = gensym() in Lam x (Succ body)) ⇝ Lam 2 (Lam 3 (Succ (Zero 2)))

- let x = gensym() in cogen (fun body -> Lam x body) ⇝
Safety in numbers

- let x = gensym() in Lam x (Zero x) ⇝ Lam 1 (Zero 1)

- let x = gensym() in Lam x
  (let body = Zero x in
   let x = gensym() in Lam x (Succ body)) ⇝ Lam 2 (Lam 3 (Succ (Zero 2)))

- let x = gensym() in cogen (fun body -> Lam x body) ⇝

Lexical scope = labels all match.

- let x = gensym() in Lam x
  (catch (let y = gensym() in Lam y
   (throw (Zero x)))) ⇝ Lam 4 (Zero 4)
Safety in numbers

let x = gensym() in Lam x (Zero x) ⇝ Lam 1 (Zero 1)

let x = gensym() in Lam x
   (let body = Zero x in
    let x = gensym() in Lam x (Succ body)) ⇝ Lam 2 (Lam 3 (Succ (Zero 2)))

let x = gensym() in cogen (fun body -> Lam x body) ⇝

Lexical scope = labels all match.

let x = gensym() in Lam x
   (catch (let y = gensym() in Lam y
           (throw (Zero y)))) ⇝ Lam 4 (Zero 5)
Meta-scope expresses binding expectations

\[ \text{uneasy } f = \text{let } x = \text{gensym()} \text{ in Lam } x \]
\[ \quad \text{(let } y = \text{gensym()} \text{ in Lam } y \]
\[ \quad \text{(f (Zero y)))} \]

\[ \text{▶ uneasy id} \xrightarrow{} \text{Lam } 6 \text{ (Lam } 7 \text{ (Zero } 7)) \]

\[ \text{▶ uneasy Succ} \xrightarrow{} \text{Lam } 6 \text{ (Lam } 7 \text{ (Succ (Zero } 7))) \]

\[ \text{▶ uneasy (fun body } \rightarrow \]
\[ \quad \text{let } z = \text{gensym()} \text{ in Lam } z \text{ (Succ body))} \xrightarrow{} \]
\[ \text{Lam } 6 \text{ (Lam } 7 \text{ (Lam } 8 \text{ (Succ (Zero } 7)))) \]

Checking easily made compositional (incremental).
Static capabilities

\[
\text{lam :: Functor } m \Rightarrow \\
(\forall s. ((\text{H Code } s \alpha, \Gamma) \rightarrow \text{Code } \alpha) \\
\quad \rightarrow m ((\text{H Code } s \alpha, \Gamma) \rightarrow \text{Code } \beta)) \\
\quad \rightarrow m (\Gamma \rightarrow \text{Code } (\alpha\rightarrow\beta))
\]

Here  \( m \)  is the effect
  \( s \)  is the static proxy for the gensym, attached using \( \text{H} \)
  \( \alpha \)  is the domain of the generated function
  \( \beta \)  is the range of the generated function
  \( \Gamma \)  is the type environment of the generated function

Claim: if the generator is well-typed, then the generated code is well-labeled.
Static capabilities

```
λm : Functor m =>
    (∀s. ((H Code s α, Γ) -> Code α)
    -> m ((H Code s α, Γ) -> Code β)
    -> m (Γ -> Code (α->β)))
```

Here  m  is the effect
    s  is the static proxy for the gensym, attached using H
    α  is the domain of the generated function
    β  is the range of the generated function
    Γ  is the type environment of the generated function

Claim: if the generator is well-typed, then the generated code is well-labeled.

▶ How to reify type-level gensym?
▶ How to unify compile-time and run-time gensym?
▶ How to automate weakening?
Waiter, there's a term in my type!

- staged lexical scope
- resource control

term gensym → type gensym
term gensym → type gensym

∀, see you later.
Lightweight monadic regions (Haskell 2008)

Goal: Resource management

- No access after close (down with run-time checking)
- Timely disposal (especially for scarce resources)
- Error handling

![Diagram showing the lifecycle of resource management with states: Open → Accessible → Close. The transition from Accessible to Access is represented by a cycle.](Diagram.png)
Type-state

test h1 = do h2 <- hOpen "config" ReadMode
  fname <- hGetLine h2
  h3 <- hOpen fname WriteMode
  hPutStrLn h3 fname
  till (liftM2 (||) (hIsEOF h2) (hIsEOF h1))
    (hGetLine h2 >>= hPutStrLn h3 >>
     hGetLine h1 >>= hPutStrLn h3)
  hClose h2
  return h3
test :: Handle -> IO Handle

test h1 = do h2 <- hOpen "config" ReadMode
            fname <- hGetLine h2
            h3 <- hOpen fname WriteMode
            hPutStrLn h3 fname
            till (liftM2 (||) (hIsEOF h2) (hIsEOF h1))
                (hGetLine h2 >>= hPutStrLn h3 >>
                 hGetLine h1 >>= hPutStrLn h3)
            hClose h2
            return h3
class Monadish m where
   return :: a -> m p p a
   (>>=) :: m p q a -> (a -> m q r b) -> m p r b

test :: STHandle 0 ->
   SIO (1,[0]) (3,[2,0]) (STHandle 2)

runSIO (... test ...)

runSIO (... hPutStrLn h3 ...)

test h1 = do h2 <- hOpen "config" ReadMode
            fname <- hGetLine h2
            h3 <- hOpen fname WriteMode
            hPutStrLn h3 fname
            till (liftM2 (||) (hIsEOF h2) (hIsEOF h1))
               (hGetLine h2 >>= hPutStrLn h3 >>=
                hGetLine h1 >>= hPutStrLn h3)
            hClose h2
            return h3
class Monadish m where
    return :: a -> m p p a
    (>>=) :: m p q a -> (a -> m q r b) -> m p r b

test :: SHandle 0 ->
    SIO (1,[0]) (3,[2,0]) (SHandle 2)

test h1 = do h2 <- hOpen "config" ReadMode
            fname <- hGetLine h2
            h3 <- hOpen fname WriteMode
            hPutStrLn h3 fname
            till (liftM2 (||) (hIsEOF h2) (hIsEOF h1))
                (hGetLine h2 >>= hPutStrLn h3 >>
                 hGetLine h1 >>= hPutStrLn h3)
            hClose h2
            return h3

do h3 <- runSIO (... test ...
             runSIO (... hPutStrLn h3 ...)

class Monadish m where
  return :: a -> m p p a
  (>>=) :: m p q a -> (a -> m q r b) -> m p r b

test :: SHandle s 0 ->
  
  SIO s (1,[0]) (3,[2,0]) (SHandle s 2)
  
  test h1 = do h2 <- hOpen "config" ReadMode
               fname <- hGetLine h2
               h3 <- hOpen fname WriteMode
               hPutStrLn h3 fname
               till (liftM2 (||) (hIsEOF h2) (hIsEOF h1))
               (hGetLine h2 >>= hPutStrLn h3 >>
                hGetLine h1 >>= hPutStrLn h3)
               hClose h2
               return h3

  do h3 <- runSIO (... test ...) 
  runSIO (... hPutStrLn h3 ...)
Waiter, there’s a term in my type!

Rank-2 polymorphism (ab)used, for safety:

- mutable state
- array index bounds
- environment classifiers
- staged lexical scope
- resource control
- automatic differentiation

runST :: (\s. ST s a) -> a

Questions:

- How to reify type-level gensym?
- How to unify compile-time and run-time gensym?
- How to compare for equality?