Computational effects across generated binders, part 2: enforcing lexical scope

Yukiyoishi Kameyama    Oleg Kiselyov    Chung-chieh Shan
University of Tsukuba    Currently Cornell

5 September 2011
Goals

Effects (error, state, let-insertion, etc.) beyond generated binders. Prevent generating

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

Effects (error, state, let-insertion, etc.) beyond generated binders. Prevent generating

- 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

- \( \text{lam} (\lambda x \rightarrow x) \rightarrow\)
  \(\text{Lam} "x1" (\text{Var} "x1")\)

- \( \text{lam} (\lambda x \rightarrow \text{let body} = x \text{ in lam} (\lambda x \rightarrow \text{body})) \rightarrow\)
  \(\text{Lam} "x2" (\text{let body} = \text{Var} "x2" \text{ in lam} (\lambda x \rightarrow \text{body})) \rightarrow\)
  \(\text{Lam} "x2" (\text{lam} (\lambda x \rightarrow \text{Var} "x2")) \rightarrow\)
  \(\text{Lam} "x2" (\text{Lam} "x3" (\text{Var} "x2"))\)

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

- \( \text{lam} (\lambda x \rightarrow \text{throw} "hello") \rightarrow\)
- \( \text{lam} (\lambda x \rightarrow \text{throw} x) \rightarrow\)
Higher-order abstract syntax

- \( \text{lam} (\ x \to \ x) \rightsquigarrow \text{Lam "x1" (Var "x1")} \)

- \( \text{lam} (\ x \to \ \text{let} \ \text{body} = x \ \text{in} \ \text{lam} (\ x \to \ \text{body})) \rightsquigarrow \text{Lam "x2" (let body = Var "x2" in lam (\ x \to \ \text{body}))} \rightsquigarrow \text{Lam "x2" (lam (\ x \to \ Var "x2"))} \rightsquigarrow \text{Lam "x2" (Lam "x3" (Var "x2"))} \)

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

- \( \text{lam} (\ x \to \ \text{throw} \ "\text{hello}" ) \rightsquigarrow ??? \)

- \( \text{lam} (\ x \to \ \text{throw} \ x ) \rightsquigarrow ??? \)

\textit{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) \rightarrow Lam "x1" (Var "x1")

- let x = gensym() in Lam x
  (let body = Var x in
   let x = gensym() in Lam x body) \rightarrow Lam "x2" (Lam "x3" (Var "x2"))

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

Ruling out scope extrusion is hard.

► let x = gensym() in Lam x (throw "hello") ⇝

► let x = gensym() in Lam x (throw (Var x)) ⇝
So, de Bruijn

- Lam Zero
- Lam (let body = Zero in Lam (Succ body)) ↝ Lam (Lam (Succ Zero))
- let x = Zero in cogen (fun body -> Lam body) ↝
So, de Bruijn

\[
\begin{align*}
\begin{align*}
\begin{align*}
\text{Lam } \text{Zero} \\
\text{Lam (let body = Zero in Lam (Succ body))} \rightsquigarrow \\
\text{Lam (Lam (Succ Zero))} \\
\text{let x = Zero in cogen (fun body -> Lam body)} \rightsquigarrow
\end{align*}
\end{align*}
\end{align*}
\]

Mourn the loss of HOAS beauty.

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

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

Open code and closed code have distinct types:

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

\[
\begin{align*}
\text{catch (Lam (throw "hello"))} &: \text{String}
\end{align*}
\]

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

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

(Kim, Yi & Calcagno, POPL 2006, §6.4)

Where did lexical scope go?
Unexpectedly bound variables

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

\[
\begin{align*}
\text{uneasy } \text{id} & \leadsto \text{Lam (Lam Zero)} \\
\text{uneasy } \text{Succ} & \leadsto \text{Lam (Lam (Succ Zero))} \\
\text{uneasy } (\text{fun } \text{body} \to \text{Lam (Succ body)}) & \leadsto \\
& \text{Lam (Lam (Lam (Succ Zero)))}
\end{align*}
\]

\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

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

\[\Rightarrow \text{uneasy id } \rightsquigarrow \text{Lam } (\text{Lam Zero})\]

\[\Rightarrow \text{uneasy Succ } \rightsquigarrow \text{Lam } (\text{Lam } (\text{Succ Zero}))\]

\[\Rightarrow \text{uneasy } (\text{fun body } \rightarrow \text{Lam } (\text{Succ body})) \rightsquigarrow \text{Lam } (\text{Lam } (\text{Lam } (\text{Succ Zero})))\]

\[\]

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})
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) \(\Rightarrow\) Lam 1 (Zero 1)

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

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

Lexical scope = labels all match.

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

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

▶ uneasy id \mapsto \text{Lam 6} \,(\text{Lam 7} \,(\text{Zero 7}))

▶ uneasy Succ \mapsto \text{Lam 6} \,(\text{Lam 7} \,(\text{Succ (Zero 7)}))

▶ uneasy (fun body -> \\
\quad \text{let } z = \text{ gensym()} \text{ in Lam } z \,(\text{Succ body})) \mapsto \\
\text{Lam 6} \,(\text{Lam 7} \,(\text{Lam 8} \,(\text{Succ (Zero 7)}))))

Checking easily made compositional (incremental).
Static capabilities

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

Here \( m \) is the effect
\( s \) is the static proxy for the gensym, attached using \( H \)
\( \alpha \) 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.

For loop tiling, \( m \) is the continuation monad for loop-insertion.
Summary

Goal:
Effects (error, state, let-insertion, etc.) beyond generated binders.
Prevent generating
- syntax errors
- type errors
- unexpectedly unbound variables
- unexpectedly bound variables

Conclusions:
Meta-types should reflect object type judgments, but that’s not enough.

Meta-bindings should reflect object bindings.
Static capabilities for early assurance.

HOAS clarity + de Bruijn flexibility.
How to improve notation? What is type-level gensym?