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

Yukiyoshi Kameyama — University of Tsukuba
Oleg Kiselyov
Chung-chieh Shan — 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

- `lam (\x -> x)` ⇝
  `Lam "x1" (Var "x1")`
- `lam (\x -> let body = x in lam (\x -> body))` ⇝
  `Lam "x2" (let body = Var "x2" in lam (\x -> body))` ⇝
  `Lam "x2" (lam (\x -> Var "x2"))` ⇝
  `Lam "x2" (Lam "x3" (Var "x2"))`

Higher-order abstract syntax

- `lam (\x -> x)` ⇝
  `Lam "x1" (Var "x1")`
- `lam (\x -> let body = x in lam (\x -> body))` ⇝
  `Lam "x2" (let body = Var "x2" in lam (\x -> body))` ⇝
  `Lam "x2" (lam (\x -> Var "x2"))` ⇝
  `Lam "x2" (Lam "x3" (Var "x2"))`

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

- `lam (\x -> throw "hello")` ⇝ ???
- `lam (\x -> throw x)` ⇝ ???

> *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 = 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))` $\leadsto$
  `Lam (Lam (Succ Zero))`
- `let x = Zero in cogen (fun body -> Lam body)` $\leadsto$

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).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{Zero} : (\Gamma, \texttt{Int} \vdash \texttt{Int})}}{\texttt{Succ Zero} : (\Gamma, \texttt{Int}, \texttt{Bool} \vdash \texttt{Int})}}{\texttt{Lam (Succ Zero)} : (\Gamma, \texttt{Int} \vdash \texttt{Bool} \to \texttt{Int})}}{\texttt{Lam (Lam (Succ Zero))} : (\Gamma \vdash \texttt{Int} \to \texttt{Bool} \to \texttt{Int})}$$

Type safety

Open code and closed code have distinct types:

$$\frac{\texttt{catch (throw (Lam Zero))} : (\vdash \texttt{Int} \to \texttt{Int})}{\texttt{run (catch (throw (Lam Zero)))} : \texttt{Int} \to \texttt{Int}}$$

$$\texttt{catch (Lam (throw "hello"))} : \texttt{String}$$

$$\texttt{catch (Lam (throw Zero))} : (\Gamma, \texttt{Int} \vdash \texttt{Int})$$

$$\frac{\dfrac{\texttt{catch (Lam (throw Zero))} : (\texttt{Int} \vdash \texttt{Int})}{\texttt{Lam (catch (Lam (throw Zero)))} : (\vdash \texttt{Int} \to \texttt{Int})}}{\texttt{run (Lam (catch (Lam (throw Zero))))} : \texttt{Int} \to \texttt{Int}}$$

(Kim, Yi & Calcagno, POPL 2006, §6.4)

Where did lexical scope go?

Unexpectedly bound variables

`uneasy f = Lam (Lam (f Zero))` (Chen & Xi, JFP 2005)

- `uneasy id` ⇝ `Lam (Lam Zero)`
- `uneasy Succ` ⇝ `Lam (Lam (Succ Zero))`
- `uneasy (fun body -> Lam (Succ body))` ⇝ `Lam (Lam (Lam (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.*
>
> *(Pouillard & Pottier, ICFP 2010)*

Unexpectedly bound variables

`uneasy f = Lam (Lam (f Zero))` (Chen & Xi, JFP 2005)

- `uneasy id` ⇝ `Lam (Lam Zero)`
- `uneasy Succ` ⇝ `Lam (Lam (Succ Zero))`
- `uneasy (fun body -> Lam (Succ body))` ⇝
  `Lam (Lam (Lam (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.
(Pouillard & Pottier, ICFP 2010)

(McBride)

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

```
uneasy f = let x = gensym() in Lam x
             (let y = gensym() in Lam y
               (f (Zero y)))
```

- `uneasy id` ⇝ `Lam 6 (Lam 7 (Zero 7))`
- `uneasy Succ` ⇝ `Lam 6 (Lam 7 (Succ (Zero 7)))`
- ```
  uneasy (fun body ->
           let z = gensym() in Lam z (Succ body)) ⇝
  Lam 6 (Lam 7 (Lam 8 (Succ (Zero 7))))
  ```

Checking easily made compositional (incremental).
  ```

Static capabilities

```
lam :: 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.

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?