A Fresh Approach to Representing Syntax with Static Binders in Functional Programming

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

(Revised version of 6 November 2001)

Functions Considered Unnecessary

Functions Considered Unnecessary for Representing Variable-Binding

A Fresh Approach to Representing Syntax with Static Binders in Functional Programming

Aims

Make the treatment of [object-level] bound variables in functional programming for syntax-manipulation (i.e. ML's original domain)

- closer to informal practice
- more declarative.

Aim: make treatment of bound variables closer to informal practice

Aim: make treatment of bound variables closer to informal practice

“Barendregt Variable Convention” (BVC)

Aim: make treatment of bound variables closer to informal practice

"Barendregt Variable Convention" (BVC)

- Operate on α-equivalence classes $[t]_\alpha$ of syntax trees via representative trees t, and
- choose names of the bound variables in t to be *fresh*, i.e. different from each other and from any free variables in the current mathematical context.

Aim: make treatment of bound variables closer to informal practice

“Barendregt Variable Convention” (BVC)

The BVC only makes sense if what we do with the representative t is *insensitive to renaming its freshly chosen bound variables* (and hence depends only on the class $[t]_\alpha$).

Aim: make treatment of bound variables closer to informal practice

“Barendregt Variable Convention” (BVC)

The BVC only makes sense if what we do with the representative $\boldsymbol{t}$ is *insensitive to renaming its freshly chosen bound variables* (and hence depends only on the class $[\boldsymbol{t}]_{\alpha}$).

Idea (Pitts & Gabbay, Proc. MPC 2000, SLNCS 1837):
Use a type system at compile-time to infer freshness properties of names that guarantee this insensitivity to renaming.

Aim: make treatment of bound variables more declarative

Aim: make treatment of bound variables more declarative

$$\left.\begin{array}{l}\text{ML's } \mathtt{\textbf{datatype}} \\ \text{Haskell's } \mathtt{\textbf{data}}\end{array}\right\}\text{ facilities}$$

reduce the task of designing data types for a given grammar's syntax trees to a mere act of declaration.

Aim: make treatment of bound variables more declarative

$$\left.\begin{array}{l}\text{ML's } \mathtt{\textbf{datatype}} \\ \text{Haskell's } \mathtt{\textbf{data}}\end{array}\right\}\text{ facilities}$$

reduce the task of designing data types for a given grammar's syntax trees to a mere act of declaration.

Can we do the same thing for syntax trees *modulo* α*-conversion of bound variables*?

Recent research provides semantic underpinnings for doing this. (Gabbay & Pitts, LICS'99; Fiore, Plotkin & Turi, LICS'99)

Grammar $\boldsymbol{term} ::=$

$$
\begin{array}{ll}
 & \boldsymbol{var} \\
\mid & \boldsymbol{term}\ \boldsymbol{term} \\
\mid & \boldsymbol{\lambda}\, \boldsymbol{var}\ .\ \boldsymbol{term} \\
\mid & \mathbf{let}\ \boldsymbol{var} = \boldsymbol{term}\ \mathbf{in}\ \boldsymbol{term} \\
\mid & \mathbf{letrec}\ \boldsymbol{var} = \boldsymbol{term}\ \mathbf{in}\ \boldsymbol{term}
\end{array}
$$

plus

specification of how $\boldsymbol{\lambda}$, $\mathbf{let}$ and $\mathbf{letrec}$ bind $\boldsymbol{var}$s (as usual)

```
datatype term =
```

$$
\begin{array}{ll}
 & \boldsymbol{var} \\
| & \boldsymbol{term}\ \boldsymbol{term} \\
| & \boldsymbol{\lambda}\ \boldsymbol{var}\ .\ \boldsymbol{term} \\
| & \mathbf{let}\ \boldsymbol{var} = \boldsymbol{term}\ \mathbf{in}\ \boldsymbol{term} \\
| & \mathbf{letrec}\ \boldsymbol{var} = \boldsymbol{term}\ \mathbf{in}\ \boldsymbol{term}
\end{array}
$$

plus

specification of how $\boldsymbol{\lambda}$, let and letrec bind $\boldsymbol{var}$s (as usual)

```
datatype term =
          Var of ν
        | term term
        | λ var . term
        | let var = term in term
        | letrec var = term in term
```

plus

specification of how $\boldsymbol{\lambda}$, $\mathbf{let}$ and $\mathbf{letrec}$ bind $\boldsymbol{var}$s (as usual)

```
datatype term =
          Var of ν
        | term term
        | λ var . term
        | let var = term in term
        | letrec var = term in term
```

ν is a type of bindable names (not **int**, **string**, ... !)

plus

specification of how λ, let and letrec bind vars (as usual)

```
datatype term =
          Var of ν
        | App of term * term
        | λ var . term
        | let var = term in term
        | letrec var = term in term
```

plus

specification of how $\boldsymbol{\lambda}$, $\mathbf{let}$ and $\mathbf{letrec}$ bind $\boldsymbol{var}$s (as usual)

```
datatype term =
          Var of ν
        | App of term * term
        | Lam of ν . term
        | let var = term in term
        | letrec var = term in term
```

plus

specification of how λ, let and letrec bind vars (as usual)

```
datatype term =
          Var of ν
        | App of term * term
        | Lam of ν . term
        | let var = term in term
        | letrec var = term in term
```

in general, ν . α is a type of name-abstractions over values of type α (not ν * α, or ν -> α !)

plus

specification of how λ, let and letrec bind vars (as usual)

```
datatype term =
            Var of ν
          | App of term * term
          | Lam of ν . term
          | Let of term * (ν . term)
          | letrec var = term in term
```

plus

specification of how λ, let and letrec bind vars (as usual)

```
datatype term =
          Var of ν
        | App of term * term
        | Lam of ν . term
        | Let of term * (ν . term)
        | Letrec of ν . (term * term)
```

plus

specification of how λ, let and letrec bind vars (as usual)

```
datatype term =
          Var of ν
        | App of term * term
        | Lam of ν . term
        | Let of term * (ν . term)
        | Letrec of ν . (term * term)
```

```
datatype term =
          Var of ν
        | App of term * term
        | Lam of ν . term
        | Let of term * (ν . term)
        | Letrec of ν . (term * term)
```

(In [Pitts & Gabbay, 2000]
ν is written as **atm** and $\nu\ .\ \alpha$ written as $[\nu]\alpha$.)

What is a type of "bindable names"?

- ν *is* like ML's `unit ref` — an equality type providing a generative supply of fresh names.

What is a type of “bindable names”?

- ν *is* like ML’s **`unit ref`** — an equality type providing a generative supply of fresh names.
- Fresh names are locally scoped via

  `fresh` ***x*** **`:`** ν **`in`** ***exp*** **`end`**

  an expression analogous to

  `let val` ***x*** **`: unit ref = ref () in`** ***exp*** **`end`**

What is a type of “bindable names”?

- ν *is* like ML’s **`unit ref`** — an equality type providing a generative supply of fresh names.
- Fresh names are locally scoped via

  `fresh` ***`x`*** **`:`** ν **`in`** ***`exp`*** **`end`**

  an expression analogous to

  `let val` ***`x`*** **`: unit ref = ref () in`** ***`exp`*** **`end`**

- The type system is used to “tame” the side-effects of dynamic name-generation. . .

ML Dynamic Semantics 101

$$\texttt{exp} \Rightarrow v$$

- $\texttt{exp}$ = expression to be evaluated
- v = semantic value of the expression

ML Dynamic Semantics 101

$$E \vdash \boldsymbol{exp} \Rightarrow v$$

- $\boldsymbol{exp}$ = expression to be evaluated
- v = semantic value of the expression
- E = environment

ML Dynamic Semantics 101

$$s, E \vdash \boldsymbol{exp} \Rightarrow v, s'$$

- $\boldsymbol{exp}$ = expression to be evaluated
- v = semantic value of the expression
- E = environment
- s = global memory state before evaluation
- s' = global memory state after evaluation

In ML, evaluation of

$$\texttt{let val } \boldsymbol{x} \texttt{ = ref() in } \boldsymbol{exp} \texttt{ end}$$

requires sequentially threaded memory states s:

$$\frac{a \notin \mathrm{dom}(s) \qquad s \cup \{a\}, E[\boldsymbol{x} \mapsto a] \vdash \boldsymbol{exp} \Rightarrow v, s'}{s, E \vdash (\texttt{let val } \boldsymbol{x} \texttt{ = ref() in } \boldsymbol{exp} \texttt{ end}) \Rightarrow v, s'}$$

In ML, evaluation of

$$\texttt{let val } \boldsymbol{x} \texttt{ = ref() in } \boldsymbol{exp} \texttt{ end}$$

requires sequentially threaded memory states s:

$$\frac{a \notin \mathrm{dom}(s) \qquad s \cup \{a\}, E[\boldsymbol{x} \mapsto a] \vdash \boldsymbol{exp} \Rightarrow v, s'}{s, E \vdash (\texttt{let val } \boldsymbol{x} \texttt{ = ref() in } \boldsymbol{exp} \texttt{ end}) \Rightarrow v, s'}$$

This has bad consequences for program calculation (e.g. function expressions no longer satisfy extensionality).

Evaluation of well-typed

$$\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}$$

requires no sequential state:

$$\frac{n \notin \mathrm{FN}(E) \qquad E[\boldsymbol{x} \mapsto n] \vdash \boldsymbol{exp} \Rightarrow v}{E \vdash (\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}) \Rightarrow v}$$

Evaluation of well-typed

$$\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}$$

requires no sequential state:

$$\frac{n \notin \mathrm{FN}(E) \qquad E[\boldsymbol{x} \mapsto n] \vdash \boldsymbol{exp} \Rightarrow v}{E \vdash (\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}) \Rightarrow v}$$

set of free names of E

Evaluation of well-typed

$$\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}$$

requires no sequential state:

$$\frac{n \notin \mathrm{FN}(E) \qquad E[\boldsymbol{x} \mapsto n] \vdash \boldsymbol{exp} \Rightarrow v}{E \vdash (\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}) \Rightarrow v}$$

type system ensures semantic invariant:
$n \notin \mathrm{FN}(E)$ implies $n \notin \mathrm{FN}(v)$

Evaluation of well-typed

$$\mathtt{fresh}\ x : \nu\ \mathtt{in}\ exp\ \mathtt{end}$$

requires no sequential state:

$$\frac{n \notin \mathrm{FN}(E) \qquad E[x \mapsto n] \vdash exp \Rightarrow v}{E \vdash (\mathtt{fresh}\ x : \nu\ \mathtt{in}\ exp\ \mathtt{end}) \Rightarrow v}$$

type system ensures semantic invariant:
$n \notin \mathrm{FN}(E)$ implies $n \notin \mathrm{FN}(v)$

Evaluation of well-typed

$$\texttt{fresh}\ x : \nu\ \texttt{in}\ exp\ \texttt{end}$$

requires no sequential state:

$$\frac{n \notin \mathrm{FN}(E) \qquad E[x \mapsto n] \vdash exp \Rightarrow v}{E \vdash (\texttt{fresh}\ x : \nu\ \texttt{in}\ exp\ \texttt{end}) \Rightarrow v}$$

Whichever name $n \notin \mathrm{FN}(E)$ is used,

type system ensures semantic invariant:
$n \notin \mathrm{FN}(E)$ implies $n \notin \mathrm{FN}(v)$

Evaluation of well-typed

$$\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}$$

requires no sequential state:

$$\frac{n \notin \mathrm{FN}(E) \qquad E[\boldsymbol{x} \mapsto n] \vdash \boldsymbol{exp} \Rightarrow v}{E \vdash (\texttt{fresh}\ \boldsymbol{x} : \nu\ \texttt{in}\ \boldsymbol{exp}\ \texttt{end}) \Rightarrow v}$$

Whichever name $n \notin \mathrm{FN}(E)$ is used, get the same v *provided the implementation identifies semantic values* v *differing only in bound names*.

Why do semantic values contain bound names?

They are introduced by evaluating
name-abstraction expressions ***x* . *exp***

Why do semantic values contain bound names?

They are introduced by evaluating name-abstraction expressions $\boldsymbol{x} \, . \, \boldsymbol{exp}$

statics:

$$\frac{\Gamma \ni (\boldsymbol{x} : \nu) \qquad \Gamma \vdash \boldsymbol{exp} : \tau}{\Gamma \vdash \boldsymbol{x} \, . \, \boldsymbol{exp} : \nu \, . \, \tau}$$

Why do semantic values contain bound names?

They are introduced by evaluating
name-abstraction expressions $\boldsymbol{x} \mathbin{.} \boldsymbol{exp}$

statics:

$$\frac{\Gamma \ni (\boldsymbol{x} : \nu) \qquad \Gamma \vdash \boldsymbol{exp} : \tau}{\Gamma \vdash \boldsymbol{x} \mathbin{.} \boldsymbol{exp} : \nu \mathbin{.} \tau}$$

dynamics:

$$\frac{E \ni (\boldsymbol{x} \mapsto n) \qquad E \vdash \boldsymbol{exp} \Rightarrow v}{E \vdash \boldsymbol{x} \mathbin{.} \boldsymbol{exp} \Rightarrow n \mathbin{.} v}$$

Why do semantic values contain bound names?

They are introduced by evaluating
name-abstraction expressions $\boldsymbol{x} \mathrel{.} \boldsymbol{exp}$

statics:

$$\frac{\Gamma \ni (\boldsymbol{x} : \nu) \qquad \Gamma \vdash \boldsymbol{exp} : \tau}{\Gamma \vdash \boldsymbol{x} \mathrel{.} \boldsymbol{exp} : \nu \mathrel{.} \tau}$$

dynamics:

$$\frac{E \ni (\boldsymbol{x} \mapsto n) \qquad E \vdash \boldsymbol{exp} \Rightarrow v}{E \vdash \boldsymbol{x} \mathrel{.} \boldsymbol{exp} \Rightarrow n \mathrel{.} v}$$

Subtle point: expression-former $\boldsymbol{x} \mathrel{.} [-]$ is *not* a binder, whereas semantic-value-former $n \mathrel{.} [-]$ is. For example. . .

x . $[-]$ is not a binder

If it were, `Lam(x . Var z)` and `Lam(y . Var z)` would be contextually equivalent— but they are not.

For example:

```
fresh x in
  fresh y in
    Lam(x . let val z = x in
      [     ]
    end)
  end
end
```

x . [−] is not a binder

If it were, ***Lam*(*x* . *Var z*)** and ***Lam*(*y* . *Var z*)** would be contextually equivalent— but they are not.

For example:

```
fresh x in
  fresh y in
    Lam(x . let val z = x in
      Lam(x . Var z)
    end)
  end
end
```

x . $[-]$ is not a binder

If it were, `Lam(x . Var z)` and `Lam(y . Var z)` would be contextually equivalent— but they are not.

For example:

```
fresh x in
  fresh y in
    Lam(x . let val z = x in
      Lam(x . Var z)
    end)
  end
end
```

evaluates to `Lam`$(n$. `Lam`$(n$. `Var` $n))$

x . $[-]$ is not a binder

If it were, ***`Lam(x . Var z)`*** and ***`Lam(y . Var z)`*** would be contextually equivalent— but they are not.

For example:

```
fresh x in
  fresh y in
    Lam(x . let val z = x in
      [     ]
    end)
  end
end
```

x . $[-]$ is not a binder

If it were, ***Lam(x . Var z)*** and ***Lam(y . Var z)*** would be contextually equivalent— but they are not.

For example:

```
fresh x in
  fresh y in
    Lam(x . let val z = x in
      Lam(y . Var z)
    end)
  end
end
```

$x \,.\, [-]$ is not a binder

If it were, **Lam(x . Var z)** and **Lam(y . Var z)** would be contextually equivalent— but they are not.

For example:

```
fresh x in
  fresh y in
    Lam(x . let val z = x in
      Lam(y . Var z)
    end)
  end
end
```

evaluates to **Lam**$(n\,.\,$**Lam**$(n'\,.\,$**Var** $n))$

How do we ensure semantic values get identified up to renaming of bound names?

Implement name-binding in the syntax of semantic values using de Bruijn indices.

How do we ensure semantic values get identified up to renaming of bound names?

Implement name-binding in the syntax of semantic values using de Bruijn indices.

This makes something automatic that was not so before:

> the language syntax provides a “nameful” interface for manipulating the general-purpose, system-level “de Bruijnery”, obviating the need for users-do-it-themselves de Bruijnery
>
> (unless they want to do it themselves for reasons of efficiency. . .).

Case-analysis of name-abstractions using pattern matching

$$\frac{E \vdash \boldsymbol{\mathtt{exp}} \Rightarrow n\,.\,v \qquad n \notin \mathrm{FN}(E) \qquad E[\boldsymbol{\mathtt{x}} \mapsto n, \boldsymbol{\mathtt{y}} \mapsto v] \vdash \boldsymbol{\mathtt{exp}}' \Rightarrow v'}{E \vdash (\mathtt{case}\ \boldsymbol{\mathtt{exp}}\ \mathtt{of}\ \boldsymbol{\mathtt{x}}\ .\ \boldsymbol{\mathtt{y}}\ \mathtt{=>}\ \boldsymbol{\mathtt{exp}}') \Rightarrow v'}$$

Case-analysis of name-abstractions using pattern matching

given n . v, can always satisfy this, because semantic values are identified up to α-equiv.

$$\frac{E \vdash \mathit{exp} \Rightarrow n \,.\, v \qquad n \notin \mathrm{FN}(E) \qquad E[\mathit{x} \mapsto n, \mathit{y} \mapsto v] \vdash \mathit{exp}' \Rightarrow v'}{E \vdash (\texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{x}\ .\ \mathit{y}\ \texttt{=>}\ \mathit{exp}') \Rightarrow v'}$$

Case-analysis of name-abstractions using pattern matching

$$\frac{E \vdash \boldsymbol{exp} \Rightarrow n \,\mathtt{.}\, v \qquad n \notin \mathrm{FN}(E) \qquad E[\boldsymbol{x} \mapsto n, \boldsymbol{y} \mapsto v] \vdash \boldsymbol{exp}' \Rightarrow v'}{E \vdash (\mathtt{case}\ \boldsymbol{exp}\ \mathtt{of}\ \boldsymbol{x}\ \mathtt{.}\ \boldsymbol{y}\ \mathtt{=>}\ \boldsymbol{exp}') \Rightarrow v'}$$

Well-typing of **`case`**

Case-analysis of name-abstractions using pattern matching

$$\frac{E \vdash \texttt{exp} \Rightarrow n \,.\, v \qquad n \notin \mathrm{FN}(E) \qquad E[\texttt{x} \mapsto n, \texttt{y} \mapsto v] \vdash \texttt{exp}' \Rightarrow v'}{E \vdash (\texttt{case exp of x . y => exp}') \Rightarrow v'}$$

Well-typing of **`case`** guarantees that the value v' is independent of the choice of name $n \notin \mathrm{FN}(E)$.

Example: capture-avoiding substitution

```
datatype term = Var of ν
              | App of term * term
              | Lam of ν . term
              | Let of term * (ν . term)
              | Letrec of ν . (term * term)
```

Example: capture-avoiding substitution

```
datatype term = Var of ν
              | App of term * term
              | Lam of ν . term
              | Let of term * (ν . term)
              | Letrec of ν . (term * term)

fun sb t x (Var y) = if x = y then t else Var y
  | sb t x (App(u , v)) = App(sb t x u , sb t x v)
  | sb t x (Lam(y . u)) = Lam(y . sb t x u)
  | sb t x (Let(u , y . v)) =
          Let(sb t x u , y . sb t x v)
  | sb t x (Letrec(y . (u , v)) =
          Letrec(y . (sb t x u , sb t x v))
```

- new forms of type

- new forms of type
- type system with “freshness inference”

- new forms of type
- type system with “freshness inference”
- bound names in semantic values…

- new forms of type
- type system with “freshness inference”
- bound names in semantic values…
- …but name-abstraction isn’t a binder

- new forms of type
- type system with “freshness inference”
- bound names in semantic values. . .
- . . . but name-abstraction isn’t a binder
- new form of pattern for name-abstraction

- new forms of type
- type system with “freshness inference”
- bound names in semantic values…
- …but name-abstraction isn’t a binder
- new form of pattern for name-abstraction

`\relax`

Key properties

- **Correctness:** α-equivalence classes of [closed] syntax trees for a grammar with binders are in bijection with [closed] values of the corresponding data type.

Key properties

- **Correctness:** α-equivalence classes of [closed] syntax trees for a grammar with binders are in bijection with [closed] values of the corresponding data type.
- **Calculation:** nice laws—because syntax-manipulation remains effect-free despite the "`gensym`-feel" of the approach.

Key properties

- **Correctness:** α-equivalence classes of [closed] syntax trees for a grammar with binders are in bijection with [closed] values of the corresponding data type.
- **Calculation:** nice laws—because syntax-manipulation remains effect-free despite the "`gensym`-feel" of the approach.
- **Convenience:** makes treatment of bound variables closer to informal practice.

Key properties

- **Correctness:** α-equivalence classes of [closed] syntax trees for a grammar with binders are in bijection with [closed] values of the corresponding data type.
- **Calculation:** nice laws—because syntax-manipulation remains effect-free despite the "`gensym`-feel" of the approach.
- **Convenience:** makes treatment of bound variables closer to informal practice.

Claim: these three **C**s are not mutually **C**ontradictory!

Key properties

- **Correctness:** α-equivalence classes of [closed] syntax trees for a grammar with binders are in bijection with [closed] values of the corresponding data type.
- **Calculation:** nice laws—because syntax-manipulation remains effect-free despite the "`gensym`-feel" of the approach.
- **Convenience:** makes treatment of bound variables closer to informal practice.

Correctness and Calculation properties established via a denotational semantics of names and name-abstraction given by *FM-sets model* (Gabbay & Pitts, LICS'99) — joint work with Gabbay & Shinwell.

Difficulties

As well as conventional typing judgements, static type system uses

> freshness judgements $\boldsymbol{x} \mathrel{\#} \boldsymbol{exp}$

whose intended meaning is

> “name bound to identifier $\boldsymbol{x}$ is not free in the semantic value to which $\boldsymbol{exp}$ evaluates (if any)”

That’s not decidable! So the static type system only gives an approximation to it.

Difficulties

- It seems hard to devise decidable freshness rules for function expressions that get very close to the intended dynamic meaning.

  (Our current freshness rule for functions is sound, but weak.)

Difficulties

- It seems hard to devise decidable freshness rules for function expressions that get very close to the intended dynamic meaning.
  (Our current freshness rule for functions is sound, but weak.)
- It's easy to go wrong, even though we have a mathematical model (FM-sets) to guide us.
  (E.g. original, "substituted-in" operational semantics was type-unsound — environment-style is OK, though.)

To do

- Try to implement this approach as an extension of a complete ML system.

  But how does freshness inference interact with polymorphism, exceptions, abstract types, references, . . . ?

To do

- Try to implement this approach as an extension of a complete ML system.

  But how does freshness inference interact with polymorphism, exceptions, abstract types, references, . . . ?
- What about a lazy version?

To do

- Try to implement this approach as an extension of a complete ML system.

  But how does freshness inference interact with polymorphism, exceptions, abstract types, references, . . . ?
- What about a lazy version?

For more information: **FreshML** project page ⟨**`www.cl.cam.ac.uk/users/amp12/freshml/`**⟩.

“Every lecture should make only one main point”

Gian-Carlo Rota

Ten Lessons I wish I Had Been Taught

Notices AMS 44(1997)22–25

“Every lecture should make only one main point”

Gian-Carlo Rota

Ten Lessons I wish I Had Been Taught

Notices AMS 44(1997)22–25

Mine is:

Familiar informal conventions about freshness of bound names in syntax-manipulating algorithms can be enforced automatically in pure functional programming via a static type system.

OUT-TAKES

Examples of typing and non-typing

```
datatype term = Var of ν
              | App of term * term
              | Lam of ν . term
              | Let of term * (ν . term)
              | Letrec of ν . (term * term)
val id = fresh x : ν in Lam(x . Var x) end
```

- *id* : *term* and *id* $\Rightarrow$ *Lam*$(n\,.\,$*Var*$\,n)$ for *any* name n
  (but note that *Lam*$(n\,.\,$*Var*$\,n) =$ *Lam*$(n'\,.\,$*Var*$\,n')$, any n, n')

Examples of typing and non-typing

```
datatype term = Var of ν
              | App of term * term
              | Lam of ν . term
              | Let of term * (ν . term)
              | Letrec of ν . (term * term)
val new_var = fresh x : ν in Var x end
```

- *new_var* is *not* well-typed.

  good! — because it evaluates non-deterministically to *Var* n, any n