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

Andrew Pitts



(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.

#### "Barendregt Variable Convention" (BVC)

#### "Barendregt Variable Convention" (BVC)

- Operate on  $\alpha$ -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.

#### "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  $[t]_{\alpha}$ ).

#### "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  $[t]_{\alpha}$ ).

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

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

```
ML's datatype
Haskell's data 
} 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*  $\alpha$ -conversion of bound variables?

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

# Grammar term ::=var| term term $| \lambda var . term$ | let var = term in term| letrec var = term in term

plus

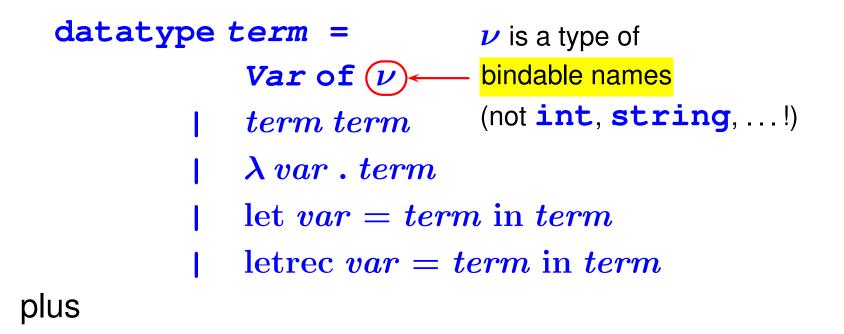
#### datatype term =

var

- term term
- $\lambda var$  . term
- let var = term in term
- letrec var = term in term

#### plus

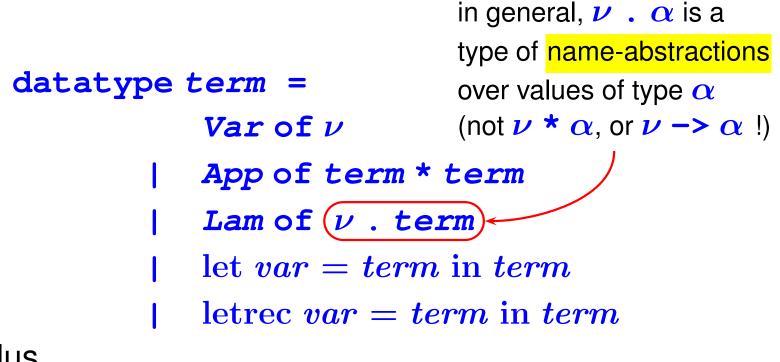
# datatype term = Var of $\nu$ | term term | $\lambda$ var . term | let var = term in term | letrec var = term in term



# datatype term = Var of $\nu$ | App of term \* term | $\lambda var \cdot term$ | let var = term in term | letrec var = term in term

# datatype term = Var of Var of App of term \* term Lam of Lam of term let var = term in term letrec var = term in term

plus



#### plus

# datatype term = Var of Var of App of term \* term Lam of term Let of term \* ( term) letrec var = term in term

#### plus

# datatype term = $Var of \nu$ | App of term \* term | Lam of $\nu$ . term | Let of term \* ( $\nu$ . term) | Letrec of $\nu$ . (term \* term)

#### plus

# datatype term = $Var of \nu$ | App of term \* term | Lam of $\nu$ . term | Let of term \* ( $\nu$ . term) | Letrec of $\nu$ . (term \* term)

# datatype term = $Var \text{ of } \nu$ | App of term \* term | Lam of $\nu$ . term | Let of term \* ( $\nu$ . term) | Letrec of $\nu$ . (term \* term)

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

## What is a type of "bindable names"?

 $\nu$  is like ML's unit ref — an equality type providing a generative supply of fresh names.

## What is a type of "bindable names"?

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

fresh  $x : \nu$  in exp end

an expression analogous to

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

## What is a type of "bindable names"?

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

```
fresh x : \nu 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**

 $exp \Rightarrow v$ 

- exp = expression to be evaluated
- v = semantic value of the expression

## **ML Dynamic Semantics 101**

 $E \vdash exp \Rightarrow v$ 

- exp = expression to be evaluated
- v = semantic value of the expression
- $\blacksquare$  **E** = environment

## **ML Dynamic Semantics 101**

 $s, E \vdash exp \Rightarrow v, s'$ 

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

In ML, evaluation of

#### let val x = ref() in exp end

requires sequentially threaded memory states s:

$$\begin{array}{l} a \notin \operatorname{dom}(s) \\ s \cup \{a\}, E[\mathbf{x} \mapsto a] \vdash e\mathbf{x}\mathbf{p} \Rightarrow v, s' \\ \hline s, E \vdash (\texttt{let val } \mathbf{x} = \texttt{ref}(\texttt{) in } e\mathbf{x}\mathbf{p} \, \texttt{end}) \Rightarrow v, s' \end{array}$$

In ML, evaluation of

#### let val x = ref() in exp end

requires sequentially threaded memory states s:

$$a \notin \operatorname{dom}(s)$$
  
 $s \cup \{a\}, E[\mathbf{x} \mapsto a] \vdash e\mathbf{x}\mathbf{p} \Rightarrow v, s'$   
 $s, E \vdash (\texttt{let val } \mathbf{x} = \texttt{ref}(\texttt{) in } e\mathbf{x}\mathbf{p} \ \texttt{end}) \Rightarrow v, s'$ 

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

Evaluation of well-typed

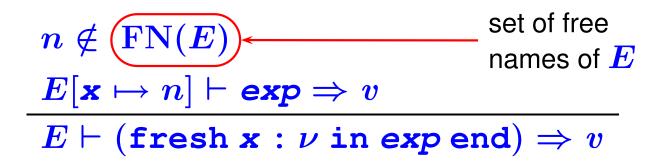
```
fresh x : v in exp end
```

requires no sequential state:

 $n \notin \operatorname{FN}(E)$  $E[\mathbf{x} \mapsto n] \vdash exp \Rightarrow v$  $E \vdash (\texttt{fresh } \mathbf{x} : \nu \texttt{ in } exp \texttt{ end}) \Rightarrow v$  Evaluation of well-typed

#### fresh x : v in exp end

requires no sequential state:



Evaluation of well-typed

```
fresh x : v in exp end
```

requires no sequential state:

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

## type system ensures semantic invariant: $n \notin FN(E)$ implies $n \notin FN(v)$ Evaluation of well-typed

#### fresh $x : \nu$ in exp end

requires no sequential state:

 $n \notin \operatorname{FN}(E)$  $E[\mathbf{x} \mapsto n] \vdash e\mathbf{x}\mathbf{p} \Rightarrow v$  $E \vdash (\operatorname{fresh} \mathbf{x} : \nu \text{ in } e\mathbf{x}\mathbf{p} \text{ end}) \Rightarrow v$ 

### type system ensures semantic invariant: $n \notin FN(E)$ implies $n \notin FN(v)$ Evaluation of well-typed

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

requires no sequential state:

$$n \notin \operatorname{FN}(E)$$
  
 $E[\mathbf{x} \mapsto n] \vdash exp \Rightarrow v$   
 $E \vdash (\operatorname{fresh} \mathbf{x} : \nu \text{ in } exp \text{ end}) \Rightarrow v$ 

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

### type system ensures semantic invariant: $n \notin FN(E)$ implies $n \notin FN(v)$ Evaluation of well-typed

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

requires no sequential state:

$$n \notin \operatorname{FN}(E)$$
  
 $E[\mathbf{x} \mapsto n] \vdash exp \Rightarrow v \longleftarrow$   
 $E \vdash (\operatorname{fresh} \mathbf{x} : \nu \text{ in } exp \text{ end}) \Rightarrow v$ 

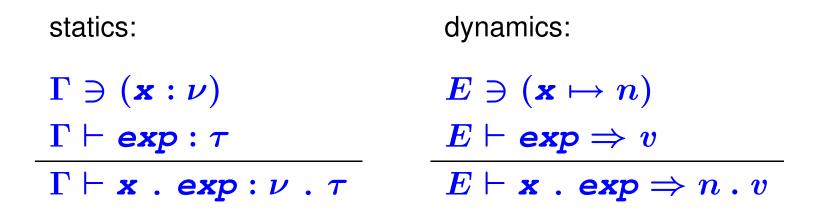
Whichever name  $n \notin FN(E)$  is used, get the same vprovided the implementation identifies semantic values v differing only in bound names.

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

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

statics:  $\Gamma \ni (\mathbf{x} : \mathbf{\nu})$   $\Gamma \vdash \mathbf{exp} : \tau$  $\Gamma \vdash \mathbf{x} \cdot \mathbf{exp} : \mathbf{\nu} \cdot \tau$ 

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



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

statics:	dynamics:
$\Gamma  i (oldsymbol{x}:oldsymbol{ u})$	$E  i (oldsymbol{x} \mapsto n)$
$\Gamma \vdash {oldsymbol exp}: au$	$E \vdash {oldsymbol exp} \Rightarrow v$
$\Gamma \vdash \mathbf{x} \cdot \mathbf{exp} : \nu \cdot \tau$	$E \vdash x \ . \ exp \Rightarrow n \ . \ v$

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

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

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

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

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

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

```
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))
```

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

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

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

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

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

```
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?

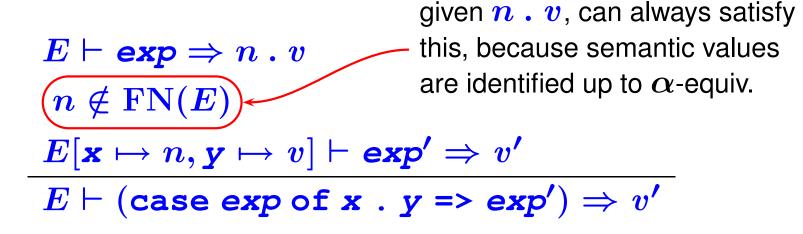
Implement name-binding in the syntax of semantic values using de Bruijn indices.

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...).

 $E \vdash exp \Rightarrow n \cdot v$   $n \notin FN(E)$   $E[x \mapsto n, y \mapsto v] \vdash exp' \Rightarrow v'$  $E \vdash (case exp of x \cdot y \Rightarrow exp') \Rightarrow v'$ 



$$\begin{array}{l} E \vdash exp \Rightarrow n \cdot v \\ n \notin FN(E) \\ \hline E[x \mapsto n, y \mapsto v] \vdash exp' \Rightarrow v' \\ \hline E \vdash (\texttt{case } exp \texttt{ of } x \cdot y \texttt{ => } exp') \Rightarrow v' \end{array}$$

$$\begin{array}{l} \text{Well-typing of } \texttt{case} \end{array}$$

$$\begin{array}{c} E \vdash exp \Rightarrow n \cdot v \\ n \notin \mathrm{FN}(E) \\ \hline E[x \mapsto n, y \mapsto v] \vdash exp' \Rightarrow v' \\ \hline E \vdash (\mathsf{case} \; exp \; \mathsf{of} \; x \cdot y \mathrel{=} \mathrel{exp'}) \Rightarrow v' \\ \hline \end{pmatrix} \\ \end{array}$$
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 \nu
| App of term * term
| Lam of \nu . term
| Let of term * (\nu . term)
| Letrec of \nu . (term * term)
```

**Example: capture-avoiding substitution** 

```
datatype term = Var of \nu
               | App of term * term
               | Lam of \nu . term
               | Let of term * (\nu . term)
               | Letrec of \nu . (term * term)
fun sbtx (Vary) = if x = y then t else Vary
  | sbtx(App(u, v)) = App(sbtxu, sbtxv)
  | sbtx(Lam(y . u)) = Lam(y . sbtxu)
  | sbtx(Let(u, y . v)) =
         Let(sbtxu, y.sbtxv)
  | sbtx(Letrec(y . (u, v)) =
         Letrec(y . (sbtxu, sbtxv))
```



- 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**

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

- 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.

- 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: 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!

- 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 x # exp

whose intended meaning is

"name bound to identifier **x** is not free in the semantic value to which **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** 

Id: term and id  $\Rightarrow$  Lam $(n \cdot Varn)$  for any name *n*(but note that Lam $(n \cdot Varn) = Lam(n' \cdot Varn')$ ,
any n, n')

**Examples of typing and non-typing** 

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

val  $new_var = fresh x : v in Var x end$ 

*new\_var* is *not* well-typed.
good! — because it evaluates non-deterministically to *Var n*, any *n*