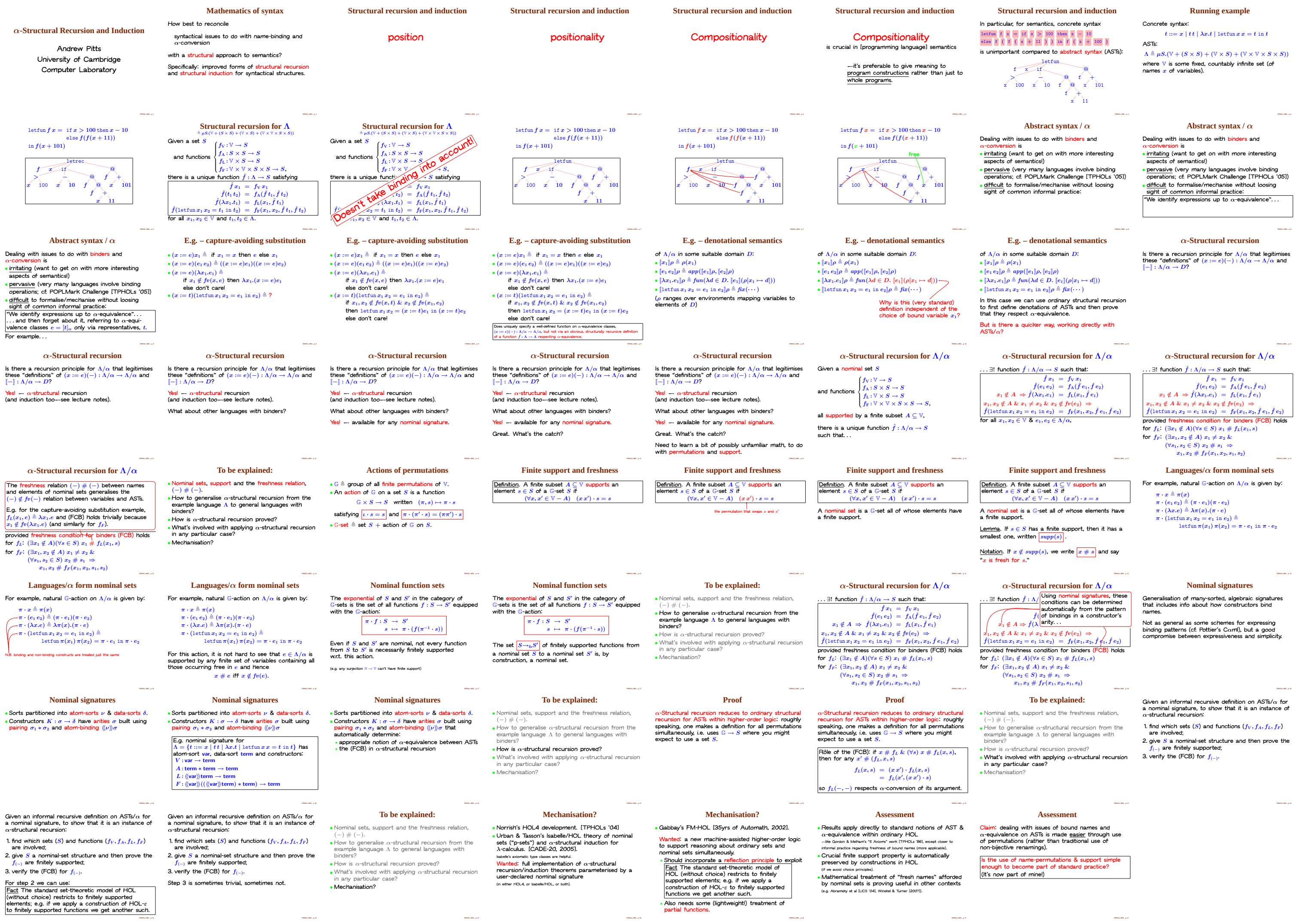
α-Structural Recursion and Induction

Andrew Pitts
University of Cambridge
Computer Laboratory

Overview

Overview



Mathematics of syntax

How best to reconcile

> syntactical issues to do with name-binding and α-conversion

with a structural approach to semantics?

Specifically: improved forms of structural recursion and structural induction for syntactical structures.

<u>Structural</u> recursion and induction

Structural recursion and induction

position

Structural recursion and induction

positionality

<u>Structural</u> recursion and induction

Compositionality

Structural recursion and induction

Compositionality

is crucial in [programming language] semantics

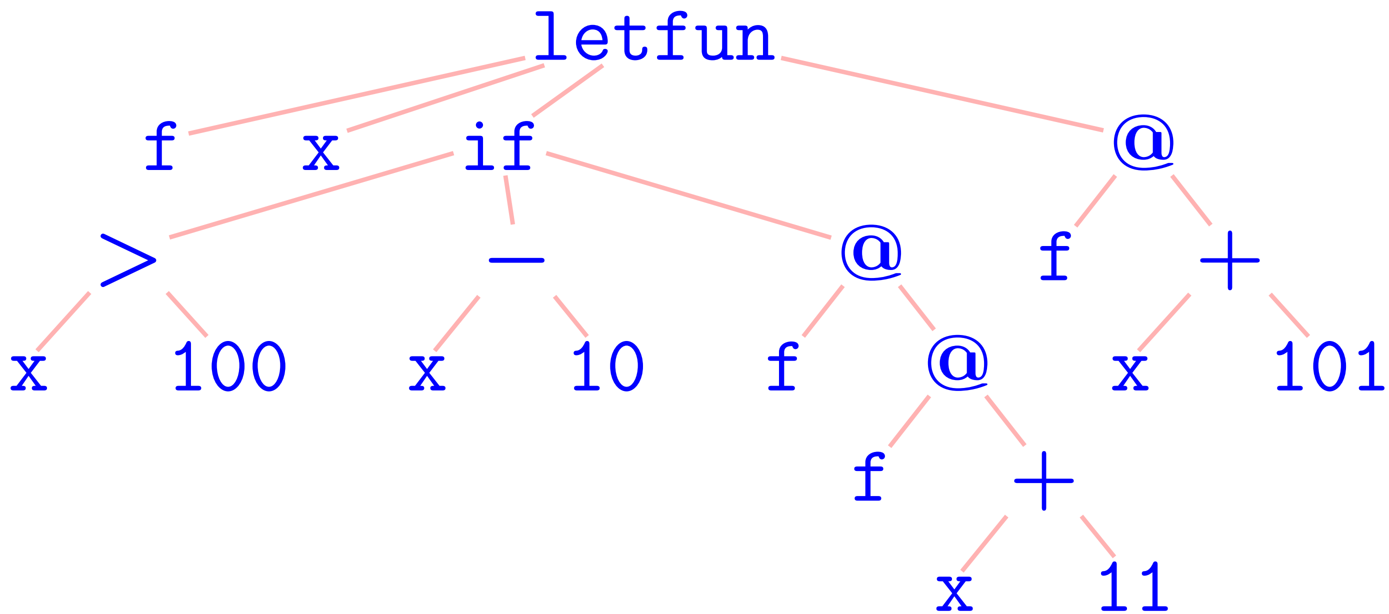
—it's preferable to give meaning to program constructions rather than just to whole programs.

Structural recursion and induction

In particular, as far as semantics is concerned, concrete syntax

`letfun` `f` `x` `=` `if` `x` `>` `100` `then` `x` `−` `10`
`else` `f` `(` `f` `(` `x` `+` `11` `)` `)` `in` `f` `(` `x` `+` `100` `)`

is unimportant compared to abstract syntax (ASTs):



Structural recursion and induction

ASTs enable two fundamental (and inter-linked) tools in programming language semantics:

- Definition of functions on syntax

  by recursion on its structure.

- Proof of properties of syntax

  by induction on its structure.

Running example

Concrete syntax:

$$t ::= x \mid t\,t \mid \lambda x.t \mid \texttt{letfun}\, x\, x = t\ \texttt{in}\ t$$

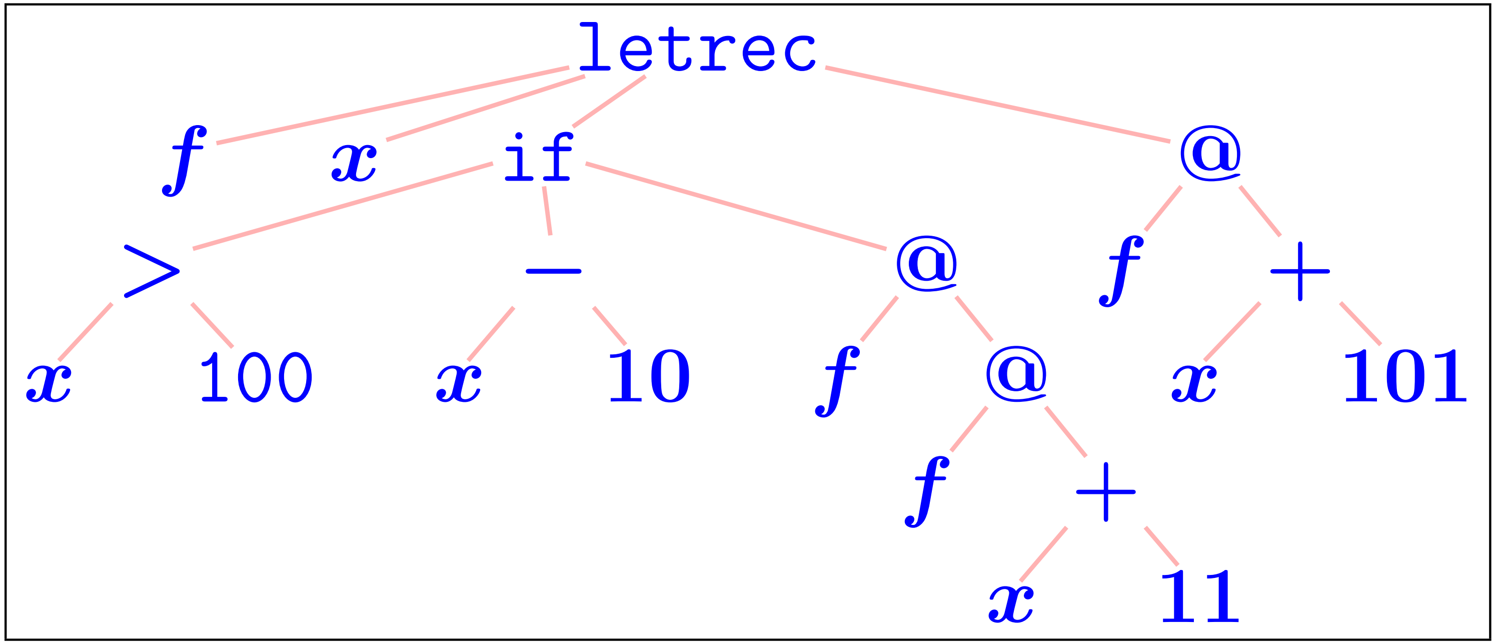
ASTs:

$$\Lambda \triangleq \mu S.(\mathbb{V} + (S \times S) + (\mathbb{V} \times S) + (\mathbb{V} \times \mathbb{V} \times S \times S))$$

where $\mathbb{V}$ is some fixed, countably infinite set (of names x of variables).

$$\begin{aligned}\texttt{letfun}\ f\ x = \ & \texttt{if}\ x > \mathbf{100}\ \texttt{then}\ x - \mathbf{10} \\ & \texttt{else}\ f(f(x+\mathbf{11}))\end{aligned}$$

$$\texttt{in}\ f(x+\mathbf{101})$$



Structural recursion for Λ

$$\triangleq \mu S.(\mathbb{V} + (S \times S) + (\mathbb{V} \times S) + (\mathbb{V} \times \mathbb{V} \times S \times S))$$

Given a set S

and functions
$$\begin{cases} f_{\mathrm{V}} : \mathbb{V} \longrightarrow S \\ f_{\mathrm{A}} : S \times S \longrightarrow S \\ f_{\mathrm{L}} : \mathbb{V} \times S \longrightarrow S \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} \times S \times S \longrightarrow S, \end{cases}$$

there is a unique function $\hat{f} : \Lambda \longrightarrow S$ satisfying

$$\boxed{\begin{aligned} \hat{f}\, x_1 &= f_{\mathrm{V}}\, x_1 \\ \hat{f}(t_1\, t_2) &= f_{\mathrm{A}}(\hat{f}\, t_1, \hat{f}\, t_2) \\ \hat{f}(\lambda x_1.t_1) &= f_{\mathrm{L}}(x_1, \hat{f}\, t_1) \\ \hat{f}(\mathtt{letfun}\, x_1\, x_2 = t_1\ \mathtt{in}\ t_2) &= f_{\mathrm{F}}(x_1, x_2, \hat{f}\, t_1, \hat{f}\, t_2) \end{aligned}}$$

for all $x_1, x_2 \in \mathbb{V}$ and $t_1, t_2 \in \Lambda$.

Structural recursion for Λ

$\triangleq \mu S.(\mathbb{V} + (S \times S) + (\mathbb{V} \times S) + (\mathbb{V} \times \mathbb{V} \times S \times S))$

Given a set S

and functions
$$\begin{cases} f_{\mathrm{V}} : \mathbb{V} \to S \\ f_{\mathrm{A}} : S \times S \to S \\ f_{\mathrm{L}} : \mathbb{V} \times S \to S \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} \times [illegible] \to S, \end{cases}$$

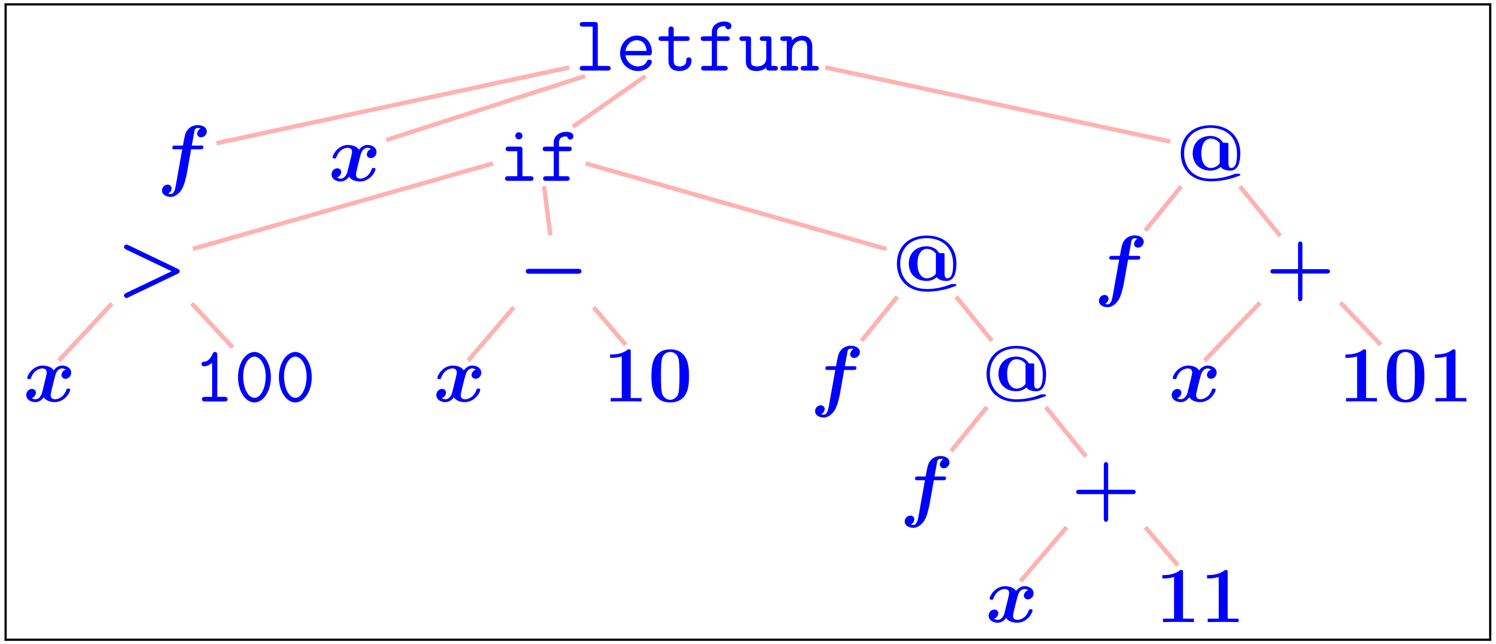
there is a unique functi[illegible] $\to S$ satisfying

$$\boxed{\begin{array}{rcl} [illegible] & = & f_{\mathrm{V}}\, x_1 \\ [illegible]\; t_2) & = & f_{\mathrm{A}}(\hat{f}\, t_1, \hat{f}\, t_2) \\ [illegible]\,(\lambda x_1.t_1) & = & f_{\mathrm{L}}(x_1, \hat{f}\, t_1) \\ \hat{f}([illegible]\, x_2 = t_1 \;\mathtt{in}\; t_2) & = & f_{\mathrm{F}}(x_1, x_2, \hat{f}\, t_1, \hat{f}\, t_2) \end{array}}$$

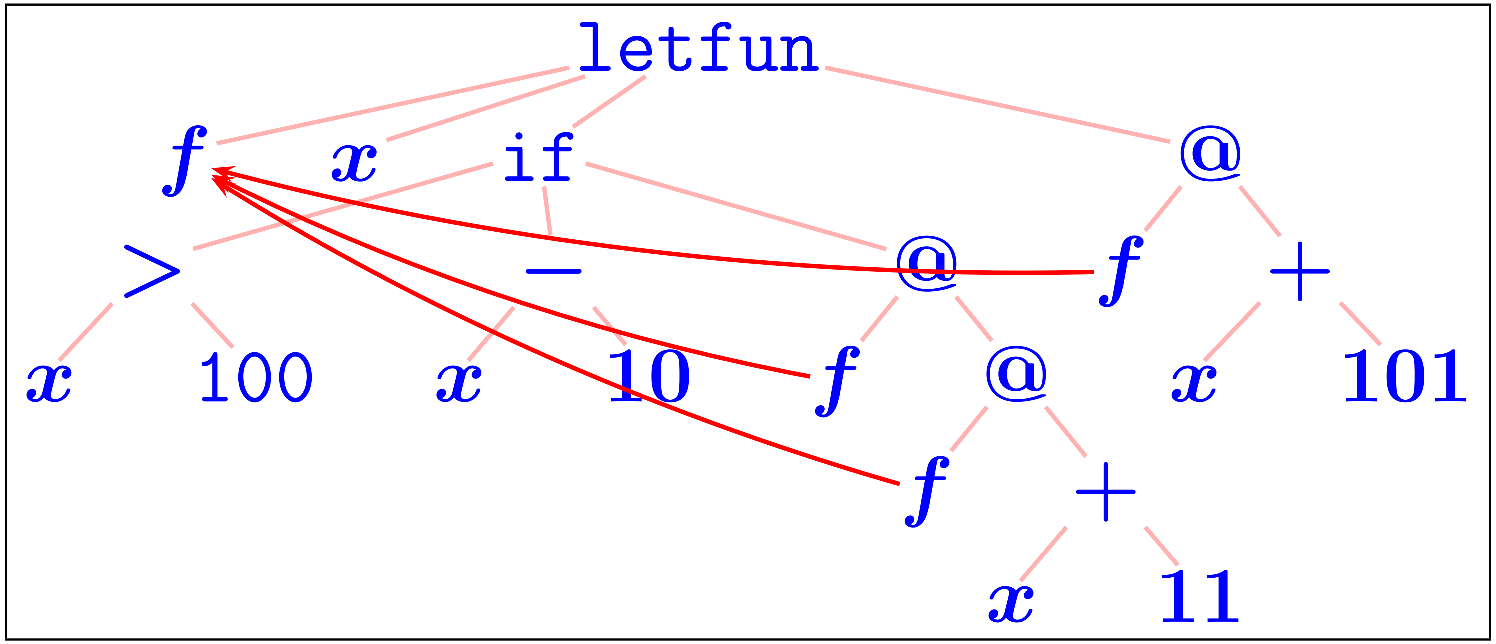
[illegible] $x_1, x_2 \in \mathbb{V}$ and $t_1, t_2 \in \Lambda$.

Doesn't take binding into account!

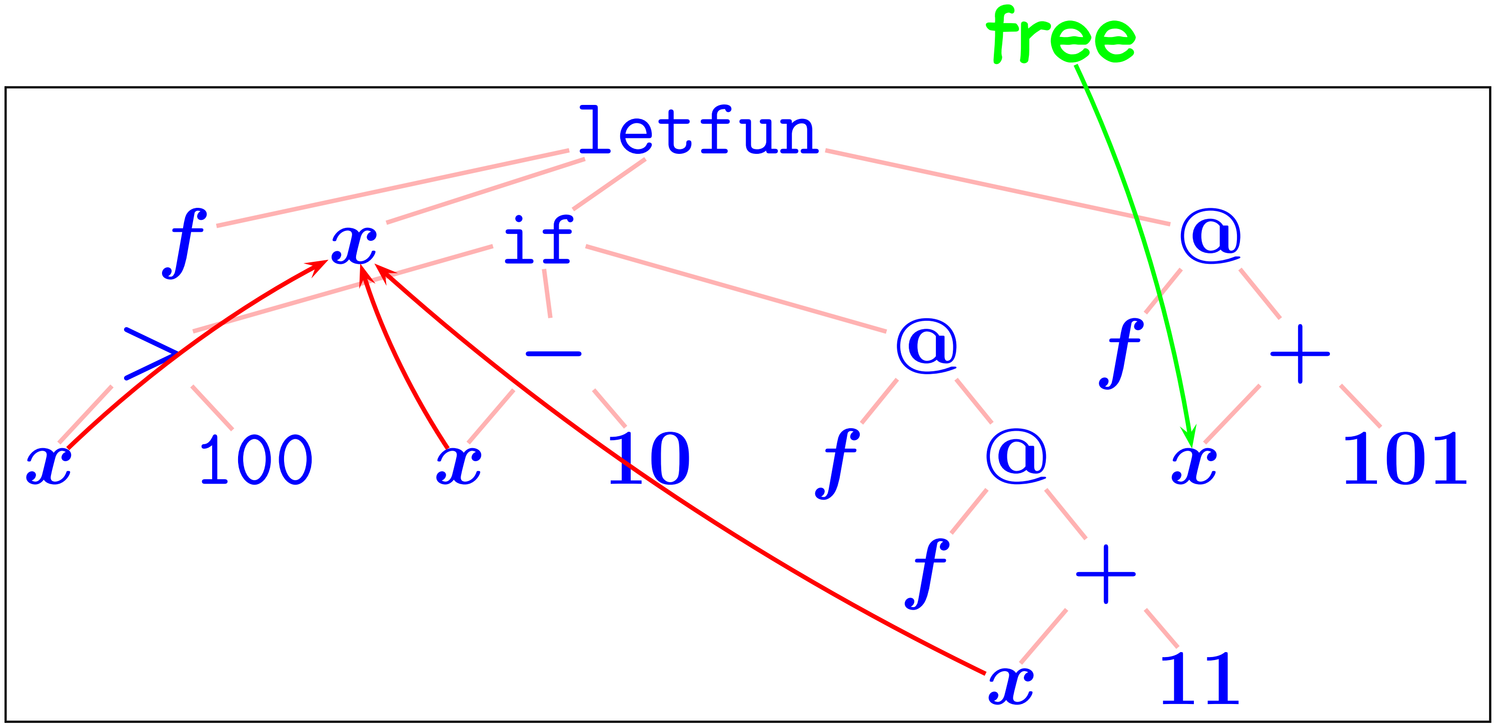
$$
\begin{aligned}
\texttt{letfun}\ f\ x = \ & \texttt{if}\ x > \mathbf{100}\ \texttt{then}\ x - \mathbf{10} \\
& \texttt{else}\ f(f(x + \mathbf{11})) \\
\texttt{in}\ f(x + \mathbf{101}) &
\end{aligned}
$$



$$\begin{aligned}
&\texttt{letfun}\ f\ x = \ \texttt{if}\ x > \mathbf{100}\ \texttt{then}\ x - \mathbf{10}\\
&\qquad\qquad\qquad\quad \texttt{else}\ f(f(x + \mathbf{11}))\\
&\texttt{in}\ f(x + \mathbf{101})
\end{aligned}$$



$$\texttt{letfun}\ f\ x = \ \texttt{if}\ x > 100\ \texttt{then}\ x - 10$$
$$\qquad\qquad\qquad\qquad \texttt{else}\ f(f(x + 11))$$
$$\texttt{in}\ f(x + 101)$$



Abstract syntax / α

Dealing with issues to do with binders and α-conversion is

- irritating (want to get on with more interesting aspects of semantics!)
- pervasive (very many languages involve binding operations; cf. POPLMark Challenge [TPHOLs '05])
- difficult to formalise/mechanise without loosing sight of common informal practice:

Abstract syntax / α

Dealing with issues to do with binders and α-conversion is

- irritating (want to get on with more interesting aspects of semantics!)
- pervasive (very many languages involve binding operations; cf. POPLMark Challenge [TPHOLs '05])
- difficult to formalise/mechanise without loosing sight of common informal practice:

"We identify expressions up to α-equivalence"...

Abstract syntax / α

Dealing with issues to do with binders and α-conversion is

- irritating (want to get on with more interesting aspects of semantics!)
- pervasive (very many languages involve binding operations; cf. POPLMark Challenge [TPHOLs '05])
- difficult to formalise/mechanise without loosing sight of common informal practice:

> "We identify expressions up to α-equivalence"...
> ...and then forget about it, referring to α-equivalence classes $e = [t]_\alpha$ only via representatives, t.

For example...

E.g. – capture-avoiding substitution

$\boxed{(x := e)e_1}$ = substitute e for all free occurrences of x in e_1, avoiding capture of free variables in e by binders in e_1.

E.g. – capture-avoiding substitution

- $(x := e)x_1 \triangleq$ if $x_1 = x$ then e else x_1
- $(x := e)(e_1\, e_2) \triangleq ((x := e)e_1)((x := e)e_2)$
- $(x := e)(\lambda x_1.e_1) \triangleq$

  if $x_1 \notin fv(x, e)$ then $\lambda x_1.(x := e)e_1$

  else don't care!
- $(x := e)(\mathtt{letfun}\, x_1\, x_2 = e_1 \,\mathtt{in}\, e_2) \triangleq$?

E.g. – capture-avoiding substitution

- $(x := e)x_1 \triangleq$ if $x_1 = x$ then e else x_1
- $(x := e)(e_1\, e_2) \triangleq ((x := e)e_1)((x := e)e_2)$
- $(x := e)(\lambda x_1.e_1) \triangleq$

  if $x_1 \notin fv(x, e)$ then $\lambda x_1.(x := e)e_1$

  else don't care!
- $(x := e)(\mathtt{letfun}\, x_1\, x_2 = e_1\ \mathtt{in}\ e_2) \triangleq$

  if $x_1, x_2 \notin fv(x, e)\ \&\ x_2 \notin fv(x_1, e_2)$

  then $\mathtt{letfun}\, x_1\, x_2 = (x := e)e_1\ \mathtt{in}\ (x := e)e_2$

  else don't care!

E.g. – capture-avoiding substitution

- $(x := e)x_1 \triangleq$ if $x_1 = x$ then e else x_1
- $(x := e)(e_1\, e_2) \triangleq ((x := e)e_1)((x := e)e_2)$
- $(x := e)(\lambda x_1.e_1) \triangleq$

  if $x_1 \notin fv(x, e)$ then $\lambda x_1.(x := e)e_1$

  else don't care!
- $(x := e)(\mathtt{letfun}\, x_1\, x_2 = e_1 \;\mathtt{in}\; e_2) \triangleq$

  if $x_1, x_2 \notin fv(x, e)$ & $x_2 \notin fv(x_1, e_2)$

  then $\mathtt{letfun}\, x_1\, x_2 = (x := e)e_1 \;\mathtt{in}\; (x := e)e_2$

  else don't care!

Does uniquely specify a well-defined function on α-equivalence classes, $(x := e)(-) : \Lambda/\alpha \to \Lambda/\alpha$, but not via an obvious, structurally recursive definition of a function $\hat{f} : \Lambda \to \Lambda$ respecting α-equivalence.

E.g. – denotational semantics

of Λ/α in some suitable domain D:

- $[\![x_1]\!]\rho \triangleq \rho(x_1)$
- $[\![e_1\,e_2]\!]\rho \triangleq \mathit{app}([\![e_1]\!]\rho, [\![e_2]\!]\rho)$
- $[\![\lambda x_1.e_1]\!]\rho \triangleq \mathit{fun}(\lambda d \in D.\ [\![e_1]\!](\rho[x_1 \mapsto d]))$
- $[\![\texttt{letfun}\, x_1\, x_2 = e_1 \texttt{ in } e_2]\!]\rho \triangleq \mathit{fix}(\cdots)$

where

- ρ ranges over environments mapping variables to elements of D
- D comes equipped with continuous functions $\mathit{app} : D \times D \to D$ and $\mathit{fun} : (D \to D) \to D$.

E.g. – denotational semantics

of Λ/α in some suitable domain D:

- $[\![x_1]\!]\rho \triangleq \rho(x_1)$
- $[\![e_1\, e_2]\!]\rho \triangleq app([\![e_1]\!]\rho, [\![e_2]\!]\rho)$
- $[\![\lambda x_1.e_1]\!]\rho \triangleq fun(\lambda d \in D.\ [\![e_1]\!](\rho[x_1 \mapsto d]))$
- $[\![\mathtt{letfun}\, x_1\, x_2 = e_1\ \mathtt{in}\ e_2]\!]\rho \triangleq fix(\cdots)$

Why is this (very standard) definition independent of the choice of bound variable x_1?

E.g. – denotational semantics

of Λ/α in some suitable domain D:

- $[\![x_1]\!]\rho \triangleq \rho(x_1)$
- $[\![e_1\, e_2]\!]\rho \triangleq app([\![e_1]\!]\rho, [\![e_2]\!]\rho)$
- $[\![\lambda x_1.e_1]\!]\rho \triangleq fun(\lambda d \in D.\, [\![e_1]\!](\rho[x_1 \mapsto d]))$
- $[\![\mathtt{letfun}\, x_1\, x_2 = e_1\ \mathtt{in}\ e_2]\!]\rho \triangleq fix(\cdots)$

In this case we can use ordinary structural recursion to first define denotations of ASTs and then prove that they respect α-equivalence.

But is there a quicker way, working directly with ASTs/α?

α-Structural recursion

Is there a recursion principle for $\boldsymbol{\Lambda/\alpha}$ that legitimises these "definitions" of $(x := e)(-) : \boldsymbol{\Lambda/\alpha} \to \boldsymbol{\Lambda/\alpha}$ and $[\![-]\!] : \boldsymbol{\Lambda/\alpha} \to \boldsymbol{D}$ (and many other e.g.s)?

α-Structural recursion

Is there a recursion principle for $\boldsymbol{\Lambda/\alpha}$ that legitimises these "definitions" of $(x := e)(-) : \boldsymbol{\Lambda/\alpha} \to \boldsymbol{\Lambda/\alpha}$ and $[\![-]\!] : \boldsymbol{\Lambda/\alpha} \to \boldsymbol{D}$ (and many other e.g.s)?

Yes! — α-structural recursion
(and induction too—see paper).

α-Structural recursion

Is there a recursion principle for $\boldsymbol{\Lambda/\alpha}$ that legitimises these "definitions" of $\boldsymbol{(x := e)(-) : \Lambda/\alpha \to \Lambda/\alpha}$ and $\boldsymbol{[\![-]\!] : \Lambda/\alpha \to D}$ (and many other e.g.s)?

Yes! — α-structural recursion
(and induction too—see paper).

What about other languages with binders?

α-Structural recursion

Is there a recursion principle for Λ/α that legitimises these "definitions" of $(x := e)(-) : \Lambda/\alpha \to \Lambda/\alpha$ and $[\![-]\!] : \Lambda/\alpha \to D$ (and many other e.g.s)?

Yes! — α-structural recursion
(and induction too—see paper).

What about other languages with binders?

Yes! — available for any nominal signature.

α-Structural recursion

Is there a recursion principle for Λ/α that legitimises these "definitions" of $(x := e)(-) : \Lambda/\alpha \to \Lambda/\alpha$ and $[\![-]\!] : \Lambda/\alpha \to D$ (and many other e.g.s)?

Yes! — α-structural recursion
(and induction too—see paper).

What about other languages with binders?

Yes! — available for any nominal signature.

Great. What's the catch?

α-Structural recursion

Is there a recursion principle for Λ/α that legitimises these "definitions" of $(x := e)(-) : \Lambda/\alpha \to \Lambda/\alpha$ and $[\![-]\!] : \Lambda/\alpha \to D$ (and many other e.g.s)?

Yes! — α-structural recursion
(and induction too—see paper).

What about other languages with binders?

Yes! — available for any nominal signature.

Great. What's the catch?

Need to learn a bit of possibly unfamiliar math, to do with permutations and support.

α-Structural recursion for Λ/α

Given a nominal set S

and functions $\begin{cases} f_{\mathrm{V}} : \mathbb{V} \rightarrow S \\ f_{\mathrm{A}} : S \times S \rightarrow S \\ f_{\mathrm{L}} : \mathbb{V} \times S \rightarrow S \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} \times S \times S \rightarrow S, \end{cases}$

all supported by a finite subset $A \subseteq \mathbb{V}$,

there is a unique function $\hat{f} : \Lambda/\alpha \rightarrow S$
such that...

α-Structural recursion for Λ/α

... $\exists!$ function $\hat{f} : \Lambda/\alpha \rightarrow S$ such that:

$$
\boxed{
\begin{array}{rcl}
\hat{f}\, x_1 & = & f_{\mathrm{V}}\, x_1 \\
\hat{f}(e_1\, e_2) & = & f_{\mathrm{A}}(\hat{f}\, e_1, \hat{f}\, e_2) \\
x_1 \notin A \;\Rightarrow\; \hat{f}(\lambda x_1.e_1) & = & f_{\mathrm{L}}(x_1, \hat{f}\, e_1) \\
\multicolumn{3}{l}{x_1, x_2 \notin A \;\&\; x_1 \neq x_2 \;\&\; x_2 \notin fv(e_2) \;\Rightarrow} \\
\hat{f}(\mathtt{letfun}\, x_1\, x_2 = e_1\; \mathtt{in}\; e_2) & = & f_{\mathrm{F}}(x_1, x_2, \hat{f}\, e_1, \hat{f}\, e_2)
\end{array}
}
$$

for all $x_1, x_2 \in \mathbb{V}$ & $e_1, e_2 \in \Lambda/\alpha$,

α-Structural recursion for Λ/α

... $\exists!$ function $\hat{f} : \Lambda/\alpha \to S$ such that:

$$\begin{array}{rcl} \hat{f}\,x_1 & = & f_{\mathrm{V}}\,x_1 \\ \hat{f}(e_1\,e_2) & = & f_{\mathrm{A}}(\hat{f}\,e_1, \hat{f}\,e_2) \\ x_1 \notin A \;\Rightarrow\; \hat{f}(\lambda x_1.e_1) & = & f_{\mathrm{L}}(x_1, \hat{f}\,e_1) \\ \multicolumn{3}{l}{x_1, x_2 \notin A \;\&\; x_1 \neq x_2 \;\&\; x_2 \notin fv(e_2) \;\Rightarrow} \\ \hat{f}(\mathtt{letfun}\,x_1\,x_2 = e_1\ \mathtt{in}\ e_2) & = & f_{\mathrm{F}}(x_1, x_2, \hat{f}\,e_1, \hat{f}\,e_2) \end{array}$$

provided freshness condition for binders (FCB) holds

for f_L: $(\exists x_1 \notin A)(\forall s \in S)\ x_1 \# f_L(x_1, s)$

for f_F: $(\exists x_1, x_2 \notin A)\ x_1 \neq x_2\ \&$

$(\forall s_1, s_2 \in S)\ x_2 \# s_1 \;\Rightarrow$

$x_1, x_2 \# f_F(x_1, x_2, s_1, s_2)$

α-Structural recursion for Λ/α

The freshness relation $(-) \# (-)$ between names and elements of nominal sets generalises the $(-) \notin fv(-)$ relation between variables and ASTs.

E.g. for the capture-avoiding substitution example, $f_L(x_1, e) \triangleq \lambda x_1.e$ and (FCB) holds trivially because $x_1 \notin fv(\lambda x_1.e)$ (and similarly for f_F).

---

provided freshness condition for binders (FCB) holds

for f_L: $(\exists x_1 \notin A)(\forall s \in S)\ x_1 \# f_L(x_1, s)$

for f_F: $(\exists x_1, x_2 \notin A)\ x_1 \neq x_2\ \&$

$$(\forall s_1, s_2 \in S)\ x_2 \# s_1 \ \Rightarrow$$

$$x_1, x_2 \# f_F(x_1, x_2, s_1, s_2)$$

To be explained:

- Nominal sets, support and the freshness relation, $(-) \# (-)$. (Simplified version of [Gabbay-Pitts, 2002].)
- How is α-structural recursion proved?
- How to generalise α-structural recursion from the example language Λ to general languages with binders?
- What's involved with applying α-structural recursion in any particular case?
- Mechanisation?

Actions of permutations

- $\mathbb{G} \triangleq$ group of all finite permutations of $\mathbb{V}$.
- An action of $\mathbb{G}$ on a set S is a function

$$\mathbb{G} \times S \to S \quad \text{written} \quad (\pi, s) \mapsto \pi \cdot s$$

satisfying $\boxed{\iota \cdot s = s}$ and $\boxed{\pi \cdot (\pi' \cdot s) = (\pi\pi') \cdot s}$

- $\mathbb{G}$-set $\triangleq$ set S + action of $\mathbb{G}$ on S.

Finite support and freshness

<u>Definition</u>. A finite subset $A \subseteq \mathbb{V}$ **supports** an element $s \in S$ of a $\mathbb{G}$-set S if

$$(\forall x, x' \in \mathbb{V} - A) \quad (x\, x') \cdot s = s$$

Finite support and freshness

Definition. A finite subset $A \subseteq \mathbb{V}$ supports an element $s \in S$ of a $\mathbb{G}$-set S if

$$(\forall x, x' \in \mathbb{V} - A) \quad (x\, x') \cdot s = s$$

the permutation that swaps x and x'

Finite support and freshness

<u>Definition</u>. A finite subset $A \subseteq \mathbb{V}$ supports an element $s \in S$ of a $\mathbb{G}$-set S if

$$(\forall x, x' \in \mathbb{V} - A) \quad (x\, x') \cdot s = s$$

A nominal set is a $\mathbb{G}$-set all of whose elements have a finite support.

Finite support and freshness

Definition. A finite subset $A \subseteq \mathbb{V}$ supports an element $s \in S$ of a $\mathbb{G}$-set S if

$$(\forall x, x' \in \mathbb{V} - A) \quad (x\,x') \cdot s = s$$

A nominal set is a $\mathbb{G}$-set all of whose elements have a finite support.

Lemma. If $s \in S$ has a finite support, then it has a smallest one, written $\boxed{supp(s)}$.

Notation. If $x \notin supp(s)$, we write $\boxed{x \# s}$ and say "x is fresh for s."

Languages/α form nominal sets

For example, natural $\mathbb{G}$-action on Λ/α is given by:

$$\pi \cdot x \triangleq \pi(x)$$
$$\pi \cdot (e_1\, e_2) \triangleq (\pi \cdot e_1)(\pi \cdot e_2)$$
$$\pi \cdot (\lambda x.e) \triangleq \lambda \pi(x).(\pi \cdot e)$$
$$\pi \cdot (\mathtt{letfun}\, x_1\, x_2 = e_1\ \mathtt{in}\ e_2) \triangleq$$
$$\mathtt{letfun}\, \pi(x_1)\, \pi(x_2) = \pi \cdot e_1\ \mathtt{in}\ \pi \cdot e_2$$

Languages/α form nominal sets

For example, natural $\mathbb{G}$-action on Λ/α is given by:

$$\pi \cdot x \triangleq \pi(x)$$
$$\pi \cdot (e_1\, e_2) \triangleq (\pi \cdot e_1)(\pi \cdot e_2)$$
$$\pi \cdot (\lambda x.e) \triangleq \lambda \pi(x).(\pi \cdot e)$$
$$\pi \cdot (\mathtt{letfun}\, x_1\, x_2 = e_1 \;\mathtt{in}\; e_2) \triangleq \mathtt{letfun}\, \pi(x_1)\, \pi(x_2) = \pi \cdot e_1 \;\mathtt{in}\; \pi \cdot e_2$$

N.B. binding and non-binding constructs are treated just the same

Languages/α form nominal sets

For example, natural $\mathbb{G}$-action on Λ/α is given by:

$$\begin{aligned}
\pi \cdot x &\triangleq \pi(x)\\
\pi \cdot (e_1\, e_2) &\triangleq (\pi \cdot e_1)(\pi \cdot e_2)\\
\pi \cdot (\lambda x.e) &\triangleq \lambda \pi(x).(\pi \cdot e)\\
\pi \cdot (\mathtt{letfun}\, x_1\, x_2 = e_1 \;\mathtt{in}\; e_2) &\triangleq\\
&\qquad \mathtt{letfun}\, \pi(x_1)\, \pi(x_2) = \pi \cdot e_1 \;\mathtt{in}\; \pi \cdot e_2
\end{aligned}$$

For this action, it is not hard to see that $e \in \Lambda/\alpha$ is supported by any finite set of variables containing all those occurring free in e and hence

$$x \# e \text{ iff } x \notin fv(e).$$

Nominal function sets

The exponential of S and S' in the category of $\mathbb{G}$-sets is the set of all functions $f : S \to S'$ equipped with the $\mathbb{G}$-action:

$$\boxed{\begin{array}{rcl} \pi \cdot f : S & \longrightarrow & S' \\ s & \longmapsto & \pi \cdot (f(\pi^{-1} \cdot s)) \end{array}}$$

With this definition, $\pi \cdot (-)$ preserves function application:

$$\begin{array}{rcl} (\pi \cdot f)(\pi \cdot s) & = & \pi \cdot (f(\pi^{-1} \cdot (\pi \cdot s))) \\ & = & \pi \cdot (f(\iota \cdot s)) \\ & = & \pi \cdot (f\, s) \end{array}$$

Nominal function sets

The exponential of S and S' in the category of $\mathbb{G}$-sets is the set of all functions $f : S \to S'$ equipped with the $\mathbb{G}$-action:

$$\boxed{\begin{array}{rcl} \pi \cdot f : S & \longrightarrow & S' \\ s & \longmapsto & \pi \cdot (f(\pi^{-1} \cdot s)) \end{array}}$$

Even if S and S' are nominal, not every function from S to S' is necessarily finitely supported w.r.t. this action.

(e.g. any surjection $\mathbb{N} \to \mathbb{V}$ can't have finite support)

Nominal function sets

The exponential of S and S' in the category of $\mathbb{G}$-sets is the set of all functions $f : S \longrightarrow S'$ equipped with the $\mathbb{G}$-action:

$$\boxed{\begin{array}{rcl} \pi \cdot f : S & \longrightarrow & S' \\ s & \longmapsto & \pi \cdot (f(\pi^{-1} \cdot s)) \end{array}}$$

The set $\boxed{S \longrightarrow_{\mathrm{fs}} S'}$ of finitely supported functions from a nominal set S to a nominal set S' is, by construction, a nominal set.

To be explained:

- Nominal sets, support and the freshness relation, $(-) \# (-)$.
  (Simplified version of [Gabbay-Pitts, 2002].)
- How is α-structural recursion proved?
- How to generalise α-structural recursion from the example language Λ to general languages with binders?
- What's involved with applying α-structural recursion in any particular case?
- Mechanisation?

Proof

α-Structural recursion reduces to ordinary structural recursion for ASTs within higher-order logic: roughly speaking, one makes a definition for all permutations simultaneously, i.e. uses $\mathbb{G} \longrightarrow S$ where you might expect to use a set S.

Proof

α-Structural recursion reduces to ordinary structural recursion for ASTs within higher-order logic: roughly speaking, one makes a definition for all permutations simultaneously, i.e. uses $\mathbb{G} \to S$ where you might expect to use a set S.

Rôle of the (FCB): if $x \mathbin{\#} f_L \;\&\; (\forall s)\; x \mathbin{\#} f_L(x, s)$, then for any $x' \mathbin{\#} (f_L, x, s)$

$$\begin{aligned} f_L(x, s) &= (x\, x') \cdot f_L(x, s) \\ &= f_L(x', (x\, x') \cdot s) \end{aligned}$$

so $f_L(-, -)$ respects α-conversion of its argument.

To be explained:

- Nominal sets, support and the freshness relation, $(-) \# (-)$.
  (Simplified version of [Gabbay-Pitts, 2002].)
- How is α-structural recursion proved?
- How to generalise α-structural recursion from the example language $\mathbf{\Lambda}$ to general languages with binders?
- What's involved with applying α-structural recursion in any particular case?
- Mechanisation?

α-Structural recursion for Λ/α

$\ldots \exists!$ function $\hat{f} : \Lambda/\alpha \to S$ such that:

$$
\boxed{
\begin{array}{rcl}
\hat{f}\, x_1 & = & f_{\mathrm{V}}\, x_1 \\
\hat{f}(e_1\, e_2) & = & f_{\mathrm{A}}(\hat{f}\, e_1, \hat{f}\, e_2) \\
x_1 \notin A \;\Rightarrow\; \hat{f}(\lambda x_1.e_1) & = & f_{\mathrm{L}}(x_1, \hat{f}\, e_1) \\
\multicolumn{3}{l}{x_1, x_2 \notin A \;\&\; x_1 \neq x_2 \;\&\; x_2 \notin fv(e_2) \;\Rightarrow} \\
\hat{f}(\mathtt{letfun}\, x_1\, x_2 = e_1 \;\mathtt{in}\; e_2) & = & f_{\mathrm{F}}(x_1, x_2, \hat{f}\, e_1, \hat{f}\, e_2)
\end{array}
}
$$

provided freshness condition for binders (FCB) holds

for f_L: $(\exists x_1 \notin A)(\forall s \in S)\; x_1 \# f_L(x_1, s)$

for f_F: $(\exists x_1, x_2 \notin A)\; x_1 \neq x_2 \;\&$

$$(\forall s_1, s_2 \in S)\; x_2 \# s_1 \;\Rightarrow$$

$$x_1, x_2 \# f_F(x_1, x_2, s_1, s_2)$$

α-Structural recursion for Λ/α

... $\exists!$ function $\hat{f} : \Lambda/$

Using nominal signatures, these conditions can be determined automatically from the pattern of bindings in a constructor's arity...

$\hat{f}($

$x_1 \notin A \Rightarrow \hat{f}(\lambda$

$x_1, x_2 \notin A \;\&\; x_1 \neq x_2 \;\&\; x_2 \notin fv(e_2) \;\Rightarrow$

$\hat{f}(\mathtt{letfun}\, x_1\, x_2 = e_1 \;\mathtt{in}\; e_2) \;=\; f_{\mathrm{F}}(x_1, x_2, \hat{f}\, e_1, \hat{f}\, e_2)$

provided freshness condition for binders (FCB) holds

for f_L: $(\exists x_1 \notin A)(\forall s \in S)\; x_1 \# f_L(x_1, s)$

for f_F: $(\exists x_1, x_2 \notin A)\; x_1 \neq x_2 \;\&$

$(\forall s_1, s_2 \in S)\; x_2 \# s_1 \;\Rightarrow$

$x_1, x_2 \# f_F(x_1, x_2, s_1, s_2)$

Nominal signatures

Generalisation of many-sorted, algebraic signatures that includes info about how constructors bind names.

Not as general as some schemes for expressing binding patterns (cf. Pottier's Cαml), but a good compromise between expressiveness and simplicity.

Nominal signatures

- Sorts partitioned into atom-sorts ν & data-sorts δ.
- Constructors $K : \sigma \longrightarrow \delta$ have arities σ built using pairing $\sigma_1 * \sigma_2$ and atom-binding $\langle\langle \nu \rangle\rangle \sigma$

Nominal signatures

- Sorts partitioned into atom-sorts ν & data-sorts δ.
- Constructors $K : \sigma \longrightarrow \delta$ have arities σ built using pairing $\sigma_1 * \sigma_2$ and atom-binding $\langle\langle \nu \rangle\rangle \sigma$

E.g. nominal signature for $\Lambda = \{t ::= x \mid t\,t \mid \lambda x.t \mid \mathtt{letfun}\,x\,x = t\ \mathtt{in}\ t\}$ has atom-sort **var**, data-sort **term** and constructors:

$V : \mathbf{var} \longrightarrow \mathbf{term}$

$A : \mathbf{term} * \mathbf{term} \longrightarrow \mathbf{term}$

$L : \langle\langle \mathbf{var} \rangle\rangle \mathbf{term} \longrightarrow \mathbf{term}$

$F : \langle\langle \mathbf{var} \rangle\rangle ((\langle\langle \mathbf{var} \rangle\rangle \mathbf{term}) * \mathbf{term}) \longrightarrow \mathbf{term}$

Nominal signatures

- Sorts partitioned into atom-sorts ν & data-sorts δ.
- Constructors $K : \sigma \longrightarrow \delta$ have arities σ built using pairing $\sigma_1 * \sigma_2$ and atom-binding $\langle\langle\nu\rangle\rangle\sigma$ that automatically determine:
  - appropriate notion of α-equivalence between ASTs
  - the (FCB) in α-structural recursion

To be explained:

- Nominal sets, support and the freshness relation, $(-) \# (-)$.
  (Simplified version of [Gabbay-Pitts, 2002].)
- How to generalise α-structural recursion from the example language Λ to general languages with binders?
- How is α-structural recursion proved?
- What's involved with applying α-structural recursion in any particular case?
- Mechanisation?

Given an informal recursive definition on ASTs/α for a nominal signature, to show that it is an instance of α-structural recursion:

1. find which sets (S) and functions (f_V, f_A, f_L, f_F) are involved;
2. give S a nominal-set structure and then prove the $f_{(-)}$ are finitely supported;
3. verify the (FCB) for $f_{(-)}$.

Given an informal recursive definition on ASTs/α for a nominal signature, to show that it is an instance of α-structural recursion:

1. find which sets (S) and functions (f_V, f_A, f_L, f_F) are involved;
2. give S a nominal-set structure and then prove the $f_{(-)}$ are finitely supported;
3. verify the (FCB) for $f_{(-)}$.

For step 2 we can use:

<u>Fact</u> The standard set-theoretic model of HOL (without choice) restricts to finitely supported elements; e.g. if we apply a construction of HOL-ε to finitely supported functions we get another such.

Given an informal recursive definition on ASTs/α for a nominal signature, to show that it is an instance of α-structural recursion:

1. find which sets (S) and functions (f_V, f_A, f_L, f_F) are involved;
2. give S a nominal-set structure and then prove the $f_{(-)}$ are finitely supported;
3. verify the (FCB) for $f_{(-)}$.

Step 3 is sometimes trivial, sometimes not.

To be explained:

- Nominal sets, support and the freshness relation, $(-) \# (-)$.
  (Simplified version of [Gabbay-Pitts, 2002].)
- How to generalise α-structural recursion from the example language Λ to general languages with binders?
- How is α-structural recursion proved?
- What's involved with applying α-structural recursion in any particular case?
- Mechanisation?

Mechanisation?

- Norrish's HOL4 development. [TPHOLs '04]
- Urban & Tasson's Isabelle/HOL theory of nominal sets ("p-sets") and α-structural induction for λ-calculus. [CADE-20, 2005].

Isabelle's axiomatic type classes are helpful.

Wanted: full implementation of α-structural recursion/induction theorems parameterised by a user-declared nominal signature

(in either HOL4, or Isabelle/HOL, or both).

Mechanisation?

- Gabbay's FM-HOL [35yrs of Automath, 2002].

  Wanted: a new machine-assisted higher-order logic to support reasoning about ordinary sets and nominal sets simultaneously.

  - Should incorporate a reflection principle to exploit

    <u>Fact</u> The standard set-theoretic model of HOL (without choice) restricts to finitely supported elements; e.g. if we apply a construction of HOL-ε to finitely supported functions we get another such.

  - Also needs some (lightweight!) treatment of partial functions.

Assessment

- Results apply directly to standard notions of AST & α-equivalence within ordinary HOL

  —like Gordon & Melham's "5 Axioms" work [TPHOLs '96], except closer to informal practice regarding freshness of bound names (more applicable).

- Crucial notion of "finite support" is automatically preserved by constructions in HOL

  (if we avoid choice principles).

- Mathematical treatment of "fresh names" afforded by nominal sets is proving useful in other contexts

  (e.g. Abramsky et al [LICS '04], Winskel & Turner [200?]).

Conclusion

Claim: dealing with issues of bound names and α-equivalence on ASTs is made <u>easier</u> through use of permutations (rather than traditional use of non-bijective renamings).

Is the use of name-permutations & support simple enough to become part of standard practice? (It's now part of mine!)