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 \( \alpha \)-conversion with a structural approach to semantics?

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

position
Structural recursion and induction

positionality
Structural 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):

```
letfun f x if > 100 then x - 10
else @ ( f ( x + 11 ) ) in f ( x + 100 )
```
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:

\[ t ::= x \mid tt \mid \lambda x.t \mid \text{letfun } x x = t \text{ in } t \]

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).
letfun \( f\ x = \) if \( x > 100 \) then \( x - 10 \) else \( f(f(x + 11)) \)
in \( f(x + 101) \)
Structural recursion for $\Lambda$

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

Given a set $S$ and functions:

- $f_V : V \rightarrow S$
- $f_A : S \times S \rightarrow S$
- $f_L : V \times S \rightarrow S$
- $f_F : V \times V \times S \times S \rightarrow S$,

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

\[
\begin{align*}
\hat{f} x_1 &= f_V x_1 \\
\hat{f}(t_1 t_2) &= f_A(\hat{f} t_1, \hat{f} t_2) \\
\hat{f}(\lambda x_1.t_1) &= f_L(x_1, \hat{f} t_1) \\
\hat{f}(\text{letfun } x_1 x_2 = t_1 \text{ in } t_2) &= f_F(x_1, x_2, \hat{f} t_1, \hat{f} t_2)
\end{align*}
\]

for all $x_1, x_2 \in V$ and $t_1, t_2 \in \Lambda$. 
Structural recursion for \( \Lambda \)

\[ \equiv \mu S. (V + (S \times S) + (V \times S) + (V \times V \times S \times S)) \]

Given a set \( S \) and functions

\[
\begin{align*}
 f_V &: V \rightarrow S \\
 f_A &: S \times S \rightarrow S \\
 f_L &: V \times S \rightarrow S \\
 f_F &: V \times V \times S \times S \rightarrow S,
\end{align*}
\]

there is a unique function \( \hat{f} : \Lambda \rightarrow S \) satisfying

\[
\begin{align*}
 \hat{f} \, x_1 &= f_V \, x_1 \\
 \hat{f} \, (\hat{f} \, t_1, \hat{f} \, t_2) &= f_A (\hat{f} \, t_1, \hat{f} \, t_2) \\
 \hat{f} \, (\lambda x_1.t_1) &= f_L (x_1, \hat{f} \, t_1) \\
 \hat{f} \, (\text{letfun } x_1 . x_2 = t_1 \text{ in } t_2) &= f_F (x_1, x_2, \hat{f} \, t_1, \hat{f} \, t_2)
\end{align*}
\]

for all \( x_1, x_2 \in V \) and \( t_1, t_2 \in \Lambda \).
letfun \( f \) \( x \) = \( \) if \( x > 100 \) then \( x - 10 \) \( \) else \( f(f(x + 11)) \) \( \) in \( f(x + 101) \)
letfun \( f \) \( x \) = if \( x > 100 \) then \( x - 10 \) else \( f(f(x + 11)) \) in \( f(x + 101) \)
\texttt{letfun}\ f\ x = \ \begin{cases} x > 100 & \text{then } x - 10 \\ \text{else} & f(f(x + 11)) \end{cases} \\
\text{in } f(x + 101)
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 / $\alpha$

Dealing with issues to do with **binders** and $\alpha$-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 $\alpha$-equivalence”...
Dealing with issues to do with binders and $\alpha$-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 losing sight of common informal practice:

"We identify expressions up to $\alpha$-equivalence"...

...and then forget about it, referring to $\alpha$-equivalence classes $e = [t]_\alpha$ only via representatives, $t$.

For example...
E.g. – capture-avoiding substitution

$$(x := e)e_1 = \text{substitute } e \text{ for all free occurrences of } x \text{ in } e_1, \text{ avoiding capture of free variables in } e \text{ by binders in } e_1.$$
E.g. – capture-avoiding substitution

- $(x := e)x_1 \triangleq \text{if } x_1 = x \text{ then } e \text{ else } x_1$
- $(x := e)(e_1 e_2) \triangleq ((x := e)e_1)((x := e)e_2)$
- $(x := e)(\lambda x_1 . e_1) \triangleq$
  
  if $x_1 \not\in \text{fv}(x, e)$ then $\lambda x_1 .(x := e)e_1$
  
  else don’t care!
- $(x := e)(\text{letfun } x_1 x_2 = e_1 \text{ in } e_2) \triangleq ?$
E.g. – capture-avoiding substitution

- $(x := e)x_1 \triangleq \text{if } x_1 = x \text{ then } e \text{ else } x_1$
- $(x := e)(e_1 e_2) \triangleq ((x := e)e_1)((x := e)e_2)$
- $(x := e)(\lambda x_1. e_1) \triangleq$
  \begin{align*}
  \text{if } x_1 \notin \text{fv}(x, e) \text{ then } & \lambda x_1.(x := e)e_1 \\
  \text{else don’t care!}
  \end{align*}
- $(x := e)(\text{letfun } x_1 x_2 = e_1 \text{ in } e_2) \triangleq$
  \begin{align*}
  \text{if } x_1, x_2 \notin \text{fv}(x, e) \& x_2 \notin \text{fv}(x_1, e_2) \\
  \text{then } \text{letfun } x_1 x_2 = (x := e)e_1 \text{ in } (x := e)e_2 \\
  \text{else don’t care!}
  \end{align*}
E.g. – capture-avoiding substitution

- \((x := e)x_1 \triangleq \text{if } x_1 = x \text{ then } e \text{ else } x_1\)
- \((x := e)(e_1 e_2) \triangleq (((x := e)e_1)((x := e)e_2))\)
- \((x := e)(\lambda x_1.e_1) \triangleq \text{if } x_1 \notin \text{fv}(x, e) \text{ then } \lambda x_1.(x := e)e_1 \text{ else don't care!}\)
- \((x := e)(\text{letfun } x_1 x_2 = e_1 \text{ in } e_2) \triangleq \text{if } x_1, x_2 \notin \text{fv}(x, e) \& x_2 \notin \text{fv}(x_1, e_2) \text{ then letfun } x_1 x_2 = (x := e)e_1 \text{ in } (x := e)e_2 \text{ else don't care!}\)

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

of $\Lambda/\alpha$ in some suitable domain $D$:

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

where

- $\rho$ ranges over environments mapping variables to elements of $D$
- $D$ comes equipped with continuous functions $\text{app} : D \times D \to D$ and
  $\text{fun} : (D \to D) \to D$. 
E.g. – denotational semantics

of $\Lambda/\alpha$ in some suitable domain $D$:

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

Why is this (very standard) definition independent of the choice of bound variable $x_1$?
E.g. – denotational semantics

of $\Lambda/\alpha$ in some suitable domain $D$:

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

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

But is there a quicker way, working directly with ASTs/$\alpha$?
α-Structural recursion

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

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

Yes! \textcolor{red}{– \textit{α-structural} recursion (and induction too—see paper).}
\textbf{\(\alpha\)-Structural recursion}

Is there a recursion principle for \(\Lambda/\alpha\) that legitimises these “definitions” of \((x := e)(\cdot) : \Lambda/\alpha \rightarrow \Lambda/\alpha\) and 
\(\llbracket \cdot \rrbracket : \Lambda/\alpha \rightarrow D\) (and many other e.g.s)?

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

What about other languages with binders?
\[\alpha\text{-Structural recursion}\]

Is there a recursion principle for \(\Lambda/\alpha\) that legitimises these “definitions” of \((x := e)(\_): \Lambda/\alpha \rightarrow \Lambda/\alpha\) and \([\_]: \Lambda/\alpha \rightarrow D\) (and many other e.g.s)?

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

What about other languages with binders?

Yes! — available for any nominal signature.
\(\alpha\)-Structural recursion

Is there a recursion principle for \(\Lambda/\alpha\) that legitimises these “definitions” of \((x := e)(\_): \Lambda/\alpha \rightarrow \Lambda/\alpha\) and \([\_]: \Lambda/\alpha \rightarrow D\) (and many other e.g.s)?

Yes! — \(\alpha\)-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 $\Lambda/\alpha$ that legitimises these “definitions” of $(x := e)(-) : \Lambda/\alpha \rightarrow \Lambda/\alpha$ and $[\cdot] : \Lambda/\alpha \rightarrow D$ (and many other e.g.s)?

Yes! — $\alpha$-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.
\(\alpha\)-Structural recursion for \(\Lambda/\alpha\)

Given a nominal set \(S\)

and functions

\[
\begin{align*}
  f_V &: \mathcal{V} \to S \\
  f_A &: S \times S \to S \\
  f_L &: \mathcal{V} \times S \to S \\
  f_F &: \mathcal{V} \times \mathcal{V} \times S \times S \to S,
\end{align*}
\]

all supported by a finite subset \(A \subseteq \mathcal{V}\),

there is a unique function \(\hat{f} : \Lambda/\alpha \to S\)

such that...
\[ \alpha\text{-Structural recursion for } \Lambda/\alpha \]

\[ \exists! \text{ function } \hat{f} : \Lambda/\alpha \to S \text{ such that:} \]

\[
\begin{align*}
\hat{f} x_1 &= f_V x_1 \\
\hat{f}(e_1 e_2) &= f_A(\hat{f} e_1, \hat{f} e_2) \\
x_1 \notin A &\Rightarrow \hat{f}(\lambda x_1.e_1) = f_L(x_1, \hat{f} e_1) \\
x_1, x_2 \notin A \land x_1 \neq x_2 \land x_2 \notin \text{fv}(e_2) &\Rightarrow \\
\hat{f}(\text{letfun } x_1 x_2 = e_1 \text{ in } e_2) &= f_F(x_1, x_2, \hat{f} e_1, \hat{f} e_2)
\end{align*}
\]

for all \( x_1, x_2 \in \forall \land e_1, e_2 \in \Lambda/\alpha, \)
**α-Structural recursion for Λ/α**

...∃! function \( \hat{f} : \Lambda/\alpha \rightarrow S \) such that:

\[
\begin{align*}
\hat{f} x_1 & = f_V x_1 \\
\hat{f}(e_1 e_2) & = f_A(\hat{f} e_1, \hat{f} e_2) \\
x_1 \notin A \Rightarrow \hat{f}(\lambda x_1.e_1) & = f_L(x_1, \hat{f} e_1) \\
x_1, x_2 \notin A & \& x_1 \neq x_2 \& x_2 \notin \text{fv}(e_2) \Rightarrow \\
\hat{f}\text{(letfun } x_1 x_2 = e_1 \text{ in } e_2) & = f_F(x_1, x_2, \hat{f} e_1, \hat{f} e_2)
\end{align*}
\]

provided freshness condition for binders (FCB) holds

for \( f_L \): \( (\exists x_1 \notin A)(\forall s \in S) \ x_1 \not\equiv f_L(x_1, s) \)

for \( f_F \): \( (\exists x_1, x_2 \notin A) \ x_1 \neq x_2 \& \\
(\forall s_1, s_2 \in S) \ x_2 \not\equiv s_1 \Rightarrow \\
x_1, x_2 \not\equiv f_F(x_1, x_2, s_1, s_2) \)
\[ \alpha \text{-Structural recursion for } \Lambda / \alpha \]

The **freshness** relation \((-) \not\# (-)\) between names and elements of nominal sets generalises the \((-) \not\in \text{fv}(-)\) 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 \(x_1 \not\in \text{fv}(\lambda x_1.e)\) (and similarly for \(f_F\)).

provided **freshness condition for binders (FCB)** holds

for \(f_L\): \((\exists x_1 \not\in A)(\forall s \in S)\) \(x_1 \not\# f_L(x_1, s)\)

for \(f_F\): \((\exists x_1, x_2 \not\in A)\) \(x_1 \neq x_2 \&\)

\((\forall s_1, s_2 \in S)\) \(x_2 \not\# s_1 \Rightarrow \)

\(x_1, x_2 \not\# f_F(x_1, x_2, s_1, s_2)\)
To be explained:

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

- $G \triangleq$ group of all finite permutations of $\mathbb{V}$.
- An action of $G$ on a set $S$ is a function $G \times S \rightarrow S$ written $(\pi, s) \mapsto \pi \cdot s$
  satisfying $\iota \cdot s = s$ and $\pi \cdot (\pi' \cdot s) = (\pi \pi') \cdot s$
- $G$-set $\triangleq$ set $S +$ action of $G$ on $S$. 
**Definition.** A finite subset $A \subseteq V$ supports an element $s \in S$ of a $G$-set $S$ if

$$(\forall x, x' \in V - A) \ (x \cdot x') \cdot s = s$$
Definition. A finite subset \( A \subseteq V \) supports an element \( s \in S \) of a \( G \)-set \( S \) if

\[
(\forall x, x' \in V - A) \quad (x \ x') \cdot s = s
\]

the permutation that swaps \( x \) and \( x' \)
Finite support and freshness

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

$$(\forall x, x' \in V - A) \ (x \cdot x') \cdot s = s$$

A nominal set is a $G$-set all of whose elements have a finite support.
Finite support and freshness

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

$$(\forall x, x' \in V - A) \ (x \cdot x') \cdot s = s$$

A nominal set is a $G$-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 $\text{supp}(s)$.

**Notation.** If $x \notin \text{supp}(s)$, we write $x \# s$ and say “$x$ is fresh for $s$.”
Languages/α form nominal sets

For example, natural $G$-action on $\Lambda/\alpha$ is given by:

\[
\begin{align*}
\pi \cdot x & \triangleq \pi(x) \\
\pi \cdot (e_1 e_2) & \triangleq (\pi \cdot e_1)(\pi \cdot e_2) \\
\pi \cdot (\lambda x. e) & \triangleq \lambda \pi(x). (\pi \cdot e) \\
\pi \cdot (\text{letfun } x_1 x_2 = e_1 \text{ in } e_2) & \triangleq \\
& \text{letfun } \pi(x_1) \pi(x_2) = \pi \cdot e_1 \text{ in } \pi \cdot e_2
\end{align*}
\]
Languages/$\alpha$ form nominal sets

For example, natural $G$-action on $\Lambda/\alpha$ is given by:

\[
\begin{align*}
\pi \cdot x & \triangleq \pi(x) \\
\pi \cdot (e_1 \ e_2) & \triangleq (\pi \cdot e_1)(\pi \cdot e_2) \\
\pi \cdot (\lambda x. e) & \triangleq \lambda \pi(x).(\pi \cdot e) \\
\pi \cdot (\text{letfun } x_1 \ x_2 = e_1 \text{ in } e_2) & \triangleq \\
& \text{letfun } \pi(x_1) \pi(x_2) = \pi \cdot e_1 \text{ in } \pi \cdot e_2
\end{align*}
\]

N.B. binding and non-binding constructs are treated just the same
Languages/\alpha form nominal sets

For example, natural \( G \)-action on \( \Lambda/\alpha \) is given by:

\[
\begin{align*}
\pi \cdot x & \triangleq \pi(x) \\
\pi \cdot (e_1 \ e_2) & \triangleq (\pi \cdot e_1)(\pi \cdot e_2) \\
\pi \cdot (\lambda x. e) & \triangleq \lambda \pi(x).(\pi \cdot e) \\
\pi \cdot (\text{letfun } x_1 \ x_2 = e_1 \text{ in } e_2) & \triangleq \\
& \quad \text{letfun } \pi(x_1) \pi(x_2) = \pi \cdot e_1 \text{ in } \pi \cdot e_2
\end{align*}
\]

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

\[
x \# e \iff x \notin \text{fv}(e).
\]
Nominal function sets

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

\[
\pi \cdot f : S \rightarrow S'
\]

\[
s \mapsto \pi \cdot (f(\pi^{-1} \cdot s))
\]

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

\[
(\pi \cdot f)(\pi \cdot s) = \pi \cdot (f(\pi^{-1} \cdot (\pi \cdot s)))
\]

\[
= \pi \cdot (f(\iota \cdot s))
\]

\[
= \pi \cdot (f \cdot s)
\]
Nominal function sets

The **exponential** of $S$ and $S'$ in the category of $G$-sets is the set of all functions $f : S \rightarrow S'$ equipped with the $G$-action:

$$\pi \cdot f : S \rightarrow S'$$

$$s \mapsto \pi \cdot (f(\pi^{-1} \cdot s))$$

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 $\mathbb{N} \rightarrow \mathbb{V}$ can’t have finite support)
Nominal function sets

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

$$
\pi \cdot f : S \rightarrow S' \\
s \mapsto \pi \cdot (f(\pi^{-1} \cdot s))
$$

The set $S \rightarrow_{fs} S'$ 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, $(-) \not\# (-)$.
  (Simplified version of [Gabbay-Pitts, 2002].)
- How is $\alpha$-structural recursion proved?
- How to generalise $\alpha$-structural recursion from the example language $\Lambda$ to general languages with binders?
- What's involved with applying $\alpha$-structural recursion in any particular case?
- Mechanisation?
Proof

\(\alpha\)-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 \(\mathbb{G} \rightarrow S\) where you might expect to use a set \(S\).
Proof

$\alpha$-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 $\mathcal{G} \rightarrow S$ where you might expect to use a set $S$.

Rôle of the (FCB): if $x \not\equiv f \land (\forall s) \; x \not\equiv f(x, s)$, then for any $x' \not\equiv (f, x, s)$

$$f(x, s) = (x \; x') \cdot f(x, s)$$
$$= f(x', (x \; x') \cdot s)$$

so $f(-, -)$ respects $\alpha$-conversion of its argument.
To be explained:

- Nominal sets, support and the freshness relation, \((-) \not\# (-)\).
  (Simplified version of [Gabbay-Pitts, 2002].)
- How is $\alpha$-structural recursion proved?
- How to generalise $\alpha$-structural recursion from the example language $\Lambda$ to general languages with binders?
- What’s involved with applying $\alpha$-structural recursion in any particular case?
- Mechanisation?
\[ \alpha\text{-Structural recursion for } \Lambda/\alpha \]

\[ \exists! \text{ function } \hat{f}: \Lambda/\alpha \rightarrow S \text{ such that:} \]

\[
\hat{f} x_1 = f_V x_1 \\
\hat{f}(e_1 e_2) = f_A(\hat{f} e_1, \hat{f} e_2) \\
x_1 \notin A \Rightarrow \hat{f}(\lambda x_1.e_1) = f_L(x_1, \hat{f} e_1) \\
x_1, x_2 \notin A \land x_1 \neq x_2 \land x_2 \notin \text{fv}(e_2) \Rightarrow \\
\hat{f}(\text{letfun } x_1 x_2 = e_1 \text{ in } e_2) = f_F(x_1, x_2, \hat{f} e_1, \hat{f} e_2)
\]

provided freshness condition for binders (FCB) holds

for \( f_L \) : \( \exists x_1 \notin A (\forall s \in S) x_1 \not\# f_L(x_1, s) \)

for \( f_F \) : \( \exists x_1, x_2 \notin A \ x_1 \neq x_2 \land \\
(\forall s_1, s_2 \in S) x_2 \not\# s_1 \Rightarrow \\
x_1, x_2 \not\# f_F(x_1, x_2, s_1, s_2) \)
\( \alpha \)-Structural recursion for \( \Lambda/\alpha \)

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

\[
\hat{f}(\lambda x_1. e_1) = f_L(x_1, \hat{f} e_1)
\]
\[
\hat{f}(e_1 e_2) = f_A(\hat{f} e_1, \hat{f} e_2)
\]
\[
x_1 \notin A \Rightarrow \hat{f}(\lambda x_1. e_1) = f_L(x_1, \hat{f} e_1)
\]

Provided freshness condition for binders (FCB) holds

For \( f_L \): \( \exists x_1 \notin A \) \( \forall s \in S \) \( x_1 \not\equiv f_L(x_1, s) \)

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

\( \forall s_1, s_2 \in S \) \( x_2 \not\equiv s_1 \Rightarrow \)

\( x_1, x_2 \not\equiv f_F(x_1, x_2, s_1, s_2) \)

Using nominal signatures, these conditions can be determined automatically from the pattern of bindings in a constructor’s arity...
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 C\(\alpha ml\)), but a good compromise between expressiveness and simplicity.
Nominal signatures

- Sorts partitioned into atom-sorts $\nu$ & data-sorts $\delta$.

- Constructors $K : \sigma \rightarrow \delta$ have arities $\sigma$ built using pairing $\sigma_1 * \sigma_2$ and atom-binding $\langle \langle \nu \rangle \rangle \sigma$
Nominal signatures

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

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

- $V : \text{var} \rightarrow \text{term}$
- $A : \text{term} \ast \text{term} \rightarrow \text{term}$
- $L : \langle\langle \text{var}\rangle\rangle \text{term} \rightarrow \text{term}$
- $F : \langle\langle \text{var}\rangle\rangle((\langle\langle \text{var}\rangle\rangle \text{term}) \ast \text{term}) \rightarrow \text{term}$
Nominal signatures

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

TPHOLs 2005, p. 23
To be explained:

- Nominal sets, support and the freshness relation, \((-) \# (-)\).
  (Simplified version of [Gabbay-Pitts, 2002].)
- How to generalise $\alpha$-structural recursion from the example language $\Lambda$ to general languages with binders?
- How is $\alpha$-structural recursion proved?
- What’s involved with applying $\alpha$-structural recursion in any particular case?
- Mechanisation?
Given an informal recursive definition on ASTs/\( \alpha \) for a nominal signature, to show that it is an instance of \( \alpha \)-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/\( \alpha \) for a nominal signature, to show that it is an instance of \( \alpha \)-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:

**Fact** The standard set-theoretic model of HOL (without choice) restricts to finitely supported elements; e.g. if we apply a construction of HOL-\( \varepsilon \) to finitely supported functions we get another such.
Given an informal recursive definition on ASTs/\(\alpha\) for a nominal signature, to show that it is an instance of \(\alpha\)-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, \((-) \not\# (-)\).
  (Simplified version of [Gabbay-Pitts, 2002].)

- How to generalise $\alpha$-structural recursion from the example language $\Lambda$ to general languages with binders?

- How is $\alpha$-structural recursion proved?

- What’s involved with applying $\alpha$-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 $\alpha$-structural induction for $\lambda$-calculus. [CADE-20, 2005].

Isabelle’s axiomatic type classes are helpful.

**Wanted**: full implementation of $\alpha$-structural recursion/induction theorems parameterised by a user-declared nominal signature

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

Fact The standard set-theoretic model of HOL (without choice) restricts to finitely supported elements; e.g. if we apply a construction of HOL-\(\varepsilon\) 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 & $\alpha$-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 $\alpha$-equivalence on ASTs is made easier 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!)