Nominal Sets and their applications

Andrew Pitts

MPhil ACS, CST Part III 2012/13
half module (8hrs)

Housekeeping

- Reading material, lecture slides and exercise sheet will be posted on the course web page.
- Assessment will be via take-home test; details *tba*.
- If you want to discuss the course material or the exercises, just send me an email, or see me at the end of a lecture.
- This course is mathematical in nature. Background knowledge is not uniform across class members and I will try to adapt to that fact. Please speak out if I use a term you do not know.

Content

Digested version of parts of three papers:

- AM Pitts, *Alpha-Structural Recursion and Induction*, JACM 53(2006)459–506.
- AM Pitts, *Structural Recursion with Locally Scoped Names*, JFP 21(2011)235–286.
- C Urban, AM Pitts and MJ Gabbay, *Nominal Unification*, TCS 323(2004) 473-497.

There is also a forthcoming book which goes into far greater detail:

- [NSB] AM Pitts, *Nominal Sets: names and symmetry in computer science* (CUP Tracts in TCS, vol. 57, 2013). Draft copies of NSB available from AMP—send request by email.

Lecture 1: introduction

Names in computer science

I'll use the term 'atomic name'

'A pure name is nothing but a bit-pattern that is an identifier, and is only useful for comparing for identity with other such bit-patterns — which includes looking up in tables to find other information. The intended contrast is with names which yield information by examination of the names themselves, whether by reading the text of the name or otherwise. ...like most good things in computer science, pure names help by putting in an extra stage of indirection; but they are not much good for anything else.'

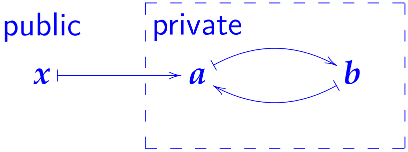
RM Needham, *Names* (ACM, 1989) p 90

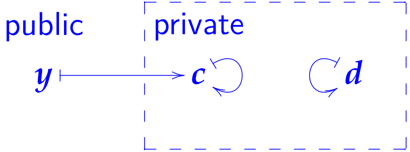
Names in computer science

Are these OCaml expressions contextually equivalent?

```
let a = ref() in
let b = ref() in
fun x →
if  x == a then b
else a
```

```
let c = ref() in
let d = ref() in
fun y →
if  y == d then d
else c
```





Names in computer science

Are these OCaml expressions contextually equivalent?

$F \triangleq$
let a = ref () in
let b = ref () in
fun x $\rightarrow$
if $x == a$ then b
else a

$G \triangleq$
let c = ref () in
let d = ref () in
fun y $\rightarrow$
if $y == d$ then d
else c

No!

For $T \triangleq$ fun $f \rightarrow$ let x = ref () in $f(f\,x) == f\,x$,

$T\,F$ has value false, whereas $T\,G$ has value true,

so $F \not\cong_{\mathbf{ctx}} G$.

Nominal sets

- Mathematical theory of names: scope, binding, freshness.
- Simple math to do with properties invariant under permuting names.
- Originally introduced by Gabbay & AMP circa 2000, but the math goes back to 1930's set theory & logic (Fraenkel & Mostowski).
- Applications: theorem-proving tools for PL semantics; metaprogramming (within functional and logic programming); verification of systems that are finite-modulo-symmetry.

Nominal sets

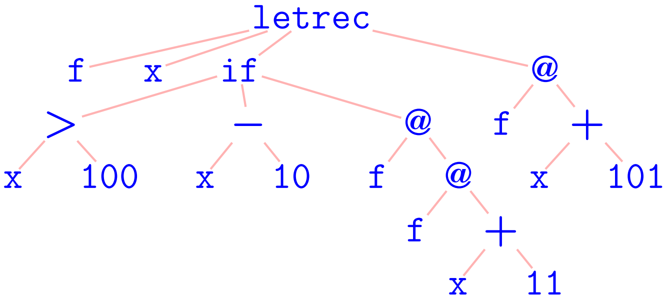
- Mathematical theory of names: scope, binding, freshness.
- Simple math to do with properties invariant under permuting names.
- Originally introduced by Gabbay & AMP circa 2000, but the math goes back to 1930's set theory & logic (Fraenkel & Mostowski).
- Applications: theorem-proving tools for PL semantics; metaprogramming (within functional and logic programming)

Motivating example: structurally recursive function definitions in the presence of name-binders.

For semantics, concrete syntax

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

is unimportant compared to abstract syntax (ASTs)



since we aim for compositional semantics of programming language constructs.

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.

Structural recursion

Recursive definitions of functions whose values at a *structure* are given functions of their values at *immediate substructures*.

- Gödel System T (1958):

$$\begin{array}{rcl} \text{structure} & = & \text{numbers} \\ \text{structural recursion} & = & \text{primitive recursion for } \mathbb{N}. \end{array}$$

- Burstall, Martin-Löf *et al* (1970s) generalised this to ASTs.

Running example

Set of ASTs for λ-terms

OCaml:

```
type vr = int;;
type tr = V of vr | A of tr * tr | L of vr * tr;;
```

Haskell:

```
type Vr = Int
data Tr = V Vr | A Tr Tr | L Vr Tr
```

Running example

Set of ASTs for λ-terms

$$Tr \triangleq \{t ::= \mathtt{V}\, a \mid \mathtt{A}(t,t) \mid \mathtt{L}(a,t)\}$$

where $a \in \mathbb{A}$, fixed infinite set of names of variables.

Operations for constructing these ASTs:

$$\begin{array}{lcl} \mathtt{V} & : & \mathbb{A} \to Tr \\ \mathtt{A} & : & Tr \times Tr \to Tr \\ \mathtt{L} & : & \mathbb{A} \times Tr \to Tr \end{array}$$

Structural recursion for Tr

Theorem.

$$\begin{array}{lrcl}
\text{Given} & f_1 & \in & \mathbb{A} \to X \\
& f_2 & \in & X \times X \to X \\
& f_3 & \in & \mathbb{A} \times X \to X \\
\hline
\text{exists unique} & \hat{f} & \in & \mathit{Tr} \to X \quad \text{satisfying}
\end{array}$$

$$\begin{aligned}
\hat{f}\,(\mathtt{V}\,a) &= f_1\,a \\
\hat{f}\,(\mathtt{A}(t,t')) &= f_2(\hat{f}\,t, \hat{f}\,t') \\
\hat{f}(\mathtt{L}(a,t)) &= f_3(a, \hat{f}\,t)
\end{aligned}$$

Structural recursion for Tr

E.g. the finite set $\mathbf{var}\,t$ of variables occurring in $t \in \mathit{Tr}$:

$$
\begin{array}{rcl}
\mathbf{var}(\mathtt{V}\,a) & = & \{a\} \\
\mathbf{var}(\mathtt{A}(t,t')) & = & (\mathbf{var}\,t) \cup (\mathbf{var}\,t') \\
\mathbf{var}(\mathtt{L}(a,t)) & = & (\mathbf{var}\,t) \cup \{a\}
\end{array}
$$

is defined by structural recursion using

- $X = \mathrm{P_f}(\mathbb{A})$ (finite sets of variables)
- $f_1\,a = \{a\}$
- $f_2(S,S') = S \cup S'$
- $f_3(a,S) = S \cup \{a\}$.

Structural recursion for Tr

E.g. swapping: $(a\ b) \cdot t =$ result of transposing all occurrences of a and b in t

For example

$$(a\ b) \cdot \mathtt{L}(a, \mathtt{A}(\mathtt{V}\, b, \mathtt{V}\, c)) = \mathtt{L}(b, \mathtt{A}(\mathtt{V}\, a, \mathtt{V}\, c))$$

Structural recursion for Tr

E.g. swapping: $(a\ b) \cdot t$ = result of transposing all occurrences of a and b in t

$$
\begin{array}{rcl}
(a\ b) \cdot \mathtt{V}\,c & = & \textbf{if } c = a \textbf{ then } \mathtt{V}\,b \textbf{ else} \\
 & & \textbf{if } c = b \textbf{ then } \mathtt{V}\,a \textbf{ else } \mathtt{V}\,c \\
(a\ b) \cdot \mathtt{A}(t, t') & = & \mathtt{A}((a\ b) \cdot t, (a\ b) \cdot t') \\
(a\ b) \cdot \mathtt{L}(c, t) & = & \textbf{if } c = a \textbf{ then } \mathtt{L}(b, (a\ b) \cdot t) \\
 & & \textbf{else if } c = b \textbf{ then } \mathtt{L}(a, (a\ b) \cdot t) \\
 & & \textbf{else } \mathtt{L}(c, (a\ b) \cdot t)
\end{array}
$$

is defined by structural recursion using. . .

Structural recursion for Tr

Theorem.

$$\begin{array}{lrcl} \text{Given} & f_1 & \in & \mathbb{A} \to X \\ & f_2 & \in & X \times X \to X \\ & f_3 & \in & \mathbb{A} \times X \to X \\ \hline \text{exists unique} & \hat{f} & \in & \mathit{Tr} \to X \quad \text{satisfying} \end{array}$$

$$\begin{array}{rcl} \hat{f}\,(\mathtt{V}\,a) & = & f_1\,a \\ \hat{f}\,(\mathtt{A}(t,t')) & = & f_2(\hat{f}\,t, \hat{f}\,t') \\ \hat{f}(\mathtt{L}(a,t)) & = & f_3(a, \hat{f}\,t) \end{array}$$

Structural recursion for *Tr*

Theorem.

Given

$$f_1 \in \mathbb{A} \to X$$
$$f_2 \in X \times \ldots$$
$$f_3 \in \ldots$$

exists unique $\hat{\ } \ldots X$ satisfying

$$\ldots = f_1\, a$$
$$\ldots t')) = f_2(\hat f\, t, \hat f\, t')$$
$$\ldots (\mathsf{L}(a,t)) = f_3(a, \hat f\, t)$$

Doesn't take binding into account!

Alpha-equivalence

Smallest binary relation $=_\alpha$ on Tr closed under the rules:

$$\frac{a \in \mathbb{A}}{\mathtt{V}\,a =_\alpha \mathtt{V}\,a} \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{\mathtt{A}(t_1, t_2) =_\alpha \mathtt{A}(t_1', t_2')}$$

$$\frac{(a\ b) \cdot t =_\alpha (a'\ b) \cdot t' \qquad b \notin \{a, a'\} \cup \mathbf{var}(t\ t')}{\mathtt{L}(a, t) =_\alpha \mathtt{L}(a', t')}$$

E.g. $\quad \mathtt{A}(\mathtt{L}(a, \mathtt{A}(\mathtt{V}\,a, \mathtt{V}\,b)), \mathtt{V}\,c) \;=_\alpha\; \mathtt{A}(\mathtt{L}(c, \mathtt{A}(\mathtt{V}\,c, \mathtt{V}\,b)), \mathtt{V}\,c)$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \neq_\alpha\; \mathtt{A}(\mathtt{L}(b, \mathtt{A}(\mathtt{V}\,b, \mathtt{V}\,b)), \mathtt{V}\,c)$

Fact: $=_\alpha$ is transitive (and reflexive & symmetric). (Exercise)

ASTs mod alpha equivalence

Dealing with issues to do with binders and alpha equivalence is

- pervasive (very many languages involve binding operations)
- difficult to formalise/mechanise without losing sight of common informal practice:

ASTs mod alpha equivalence

Dealing with issues to do with binders and alpha equivalence is

- pervasive (very many languages involve binding operations)
- difficult to formalise/mechanise without losing sight of common informal practice:

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

ASTs mod alpha equivalence

Dealing with issues to do with binders and alpha equivalence is

- pervasive (very many languages involve binding operations)
- 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 $[t]_\alpha$ only via representatives t.

ASTs mod alpha equivalence

Dealing with issues to do with binders and alpha equivalence is

- pervasive (very many languages involve binding operations)
- difficult to formalise/mechanise without losing sight of common informal practice:

E.g. notation for λ-terms:

$$\Lambda \triangleq \{[t]_\alpha \mid t \in Tr\}$$

$$\begin{array}{rll} a & \text{means} & [\mathtt{V}\,a]_\alpha \ (=\{\mathtt{V}\,a\}) \\ e\,e' & \text{means} & [\mathtt{A}(t,t')]_\alpha, \text{ where } e=[t]_\alpha \text{ and } e'=[t']_\alpha \\ \lambda a.e & \text{means} & [\mathtt{L}(a,t)]_\alpha \text{ where } e=[t]_\alpha \end{array}$$

Informal structural recursion

E.g. capture-avoiding substitution:

$$f = (-)[e_1/a_1] : \Lambda \to \Lambda$$

$$\begin{aligned}
f\,a &= \textbf{if } a = a_1 \textbf{ then } e_1 \textbf{ else } a \\
f\,(e\,e') &= (f\,e)\,(f\,e') \\
f(\lambda a.e) &= \textbf{if } a \notin \mathbf{fv}(a_1, e_1) \textbf{ then } \lambda a.(f\,e) \\
&\qquad \textbf{else } \text{don't care!}
\end{aligned}$$

Not an instance of structural recursion for Tr.

Why is f well-defined and total?

Informal structural recursion

E.g. denotation of λ-term in a suitable domain D:

$$[\![-]\!] : \Lambda \to ((\mathbb{A} \to D) \to D)$$

$$\begin{array}{rcl}
[\![a]\!]\rho & = & \rho\, a \\
[\![e\, e']\!]\rho & = & app([\![e]\!]\rho\,, [\![e']\!]\rho) \\
[\![\lambda a.\, e]\!]\rho & = & fun(\lambda(d \in D).\, [\![e]\!](\rho[a \to d]))
\end{array}$$

where $\begin{cases} app \in D \times D \to_{cts} D \\ fun \in (D \to_{cts} D) \to_{cts} D \end{cases}$

are continuous functions satisfying. . .

Informal structural recursion

E.g. denotation of λ-term in a suitable domain D:

$$[\![-]\!] : \Lambda \to ((\mathbb{A} \to D) \to D)$$

$$
\begin{aligned}
[\![a]\!]\rho &= \rho\, a \\
[\![e\, e']\!]\rho &= app([\![e]\!]\rho\,,[\![e']\!]\rho) \\
[\![\lambda a.\, e]\!]\rho &= fun(\lambda(d \in D).\,[\![e]\!](\rho[a \to d]))
\end{aligned}
$$

why is this very standard definition independent of the choice of bound variable a?

Is there a recursion principle for $\mathbf{\Lambda}$ that legitimises these 'definitions' of $(-)[e_1/a_1] : \mathbf{\Lambda} \to \mathbf{\Lambda}$ and $[\![-]\!] : \mathbf{\Lambda} \to D$ (and many other e.g.s)?

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

What about other languages with binders?

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

What about other languages with binders?

Yes! — available for any nominal signature.

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

What about other languages with binders?

Yes! — available for any nominal signature.

Great. What's the catch?

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

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.