Nominal System T

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Primitive Recursion: recursive definitions of (total) functions where value at a *structure* is a given function of its value at *immediate substructures*.

- Gödel (Tate) System T — structure = numbers.
- Burstall, Martin-Löf *et al* generalized this to abstract syntax trees.

Primitive Recursion: recursive definitions of (total) functions where value at a *structure* is a given function of its value at *immediate substructures*.

- Gödel (Tate) System T — structure = numbers.
- Burstall, Martin-Löf *et al* generalized this to abstract syntax trees.
- **Nominal System T** (NST): generalizes to abstract syntax trees quotiented by α-equivalence via Odersky-style local names + name-permutations.

NST formalizes common practice with bound names, e.g. . .

λ-terms $t = \lambda$-trees mod α-equivalence

$a \mapsto \mathtt{V}\,a$	variables
$t, t' \mapsto \mathtt{A}\,t\,t'$	application terms
$a, t \mapsto \mathtt{L}\,a.\,t$	λ-abstraction terms

Typical e.g. of "not quite" primitive recursion:

$$f = (-)[t_1/a_1]$$

(capture-avoiding substitution)

is well-(and totally-)defined by:

$$\begin{aligned} f\,(\mathtt{V}\,a) &= \textbf{if } a = a_1 \textbf{ then } t_1 \textbf{ else } \mathtt{V}\,a \\ f\,(\mathtt{A}\,t\,t') &= \mathtt{A}\,(f\,t)\,(f\,t') \\ f(\mathtt{L}\,a.\,t) &= \mathtt{L}\,a.\,(f\,t) \quad \text{if } a \mathbin{\#} a_1, t_1 \end{aligned}$$

In general:

$$
\begin{array}{rcll}
f\,(\mathtt{V}\,a) & = & f_1\,a & \\
f\,(\mathtt{A}\,t\,t') & = & f_2\,t\,t'\,(f\,t)\,(f\,t') & \\
f\,(\mathtt{L}\,a.\,t) & = & f_3\,a\,t\,(f\,t) & \text{if } a \mathrel{\#} f_1, f_2, f_2
\end{array}
$$

In general:

$$
\begin{aligned}
f\,(\mathtt{V}\,a) &= f_1\,a \\
f\,(\mathtt{A}\,t\,t') &= f_2\,t\,t'\,(f\,t)\,(f\,t') \\
f(\mathtt{L}\,a.\,t) &= f_3\,a\,t\,(f\,t) \qquad \text{if } a \,\#\, f_1, f_2, f_2
\end{aligned}
$$

$= \mathtt{L}\,a'.\,t'$ $\qquad$ $= f_3\,a'\,t'\,(f\,t')$

Q: how to get rid of this inconvenient proof obligation?

In general:

$$
\begin{aligned}
f\,(\mathtt{V}\,a) &= f_1\,a \\
f\,(\mathtt{A}\,t\,t') &= f_2\,t\,t'\,(f\,t)\,(f\,t') \\
f(\mathtt{L}\,a.\,t) &= \nu a.\, f_3\,a\,t\,(f\,t) \qquad [\,a \# f_1, f_2, f_2\,]
\end{aligned}
$$

$= \mathtt{L}\,a'.\,t'$ $\qquad = \nu a'.\, f_3\,a'\,t'\,(f\,t')$ ***OK!***

Q: how to get rid of this inconvenient proof obligation?

A: use a local scoping construct $\nu a.\,(-)$ for names

In general:

$$\begin{aligned} f\,(\mathtt{V}\,a) &= f_1\,a \\ f\,(\mathtt{A}\,t\,t') &= f_2\,t\,t'\,(f\,t)\,(f\,t') \\ f\,(\mathtt{L}\,a.\,t) &= \nu a.\,f_3\,a\,t\,(f\,t) \qquad [\,a \mathbin{\#} f_1, f_2, f_2\,] \end{aligned}$$

A: use a local scoping construct $\nu a.\,(-)$ for names

which one?!

Dynamic allocation

- Familiar—widely used in practice.
- Stateful: $\nu a.\, e$ means "add a fresh name a' to the current state and return $e[a'/a]$"

Dynamic allocation

- Familiar—widely used in practice.
- Stateful: $\nu a.\,e$ means "add a fresh name a' to the current state and return $e[a'/a]$"
- Disrupts familiar mathematical properties of pure datatypes. E.g. tuples do not behave extensionally:

$$\begin{array}{lrcl} & \mathtt{fst}(\nu a\, a'.\,(a\,,a')) & \approx & \mathtt{fst}(\nu a.\,(a\,,a)) \\ \text{and} & \mathtt{snd}(\nu a\, a'.\,(a\,,a')) & \approx & \mathtt{snd}(\nu a.\,(a\,,a)) \\ \hline \text{but} & \nu a\, a'.\,(a\,,a') & \not\approx & \nu a.\,(a\,,a) \end{array}$$

(consider $\mathtt{case}\ [-]\ \mathtt{of}\ (x\,,x') \to x = x'$)

(and similar nasties for functions).

So we reject it in favour of...

Odersky's $\nu a.(-)$ [POPL'94]

- Unfamiliar—apparently not used in practice (so far).
- Pure equational calculus.
- Tuples and functions obey familiar mathematical laws, because

$$\begin{aligned} \nu a.(\lambda x.e) &\approx \lambda x.(\nu a.e) \\ \nu a.(e, e') &\approx (\nu a.e, \nu a.e') \end{aligned}$$

  so e.g. unlike for dynamic allocation, one has

$$\begin{aligned} \nu a\, a'.(a, a') &\approx (\nu a\, a'.a, \nu a\, a'.a') \\ &\approx (\nu a.a, \nu a.a) \\ &\approx \nu a.(a, a) \end{aligned}$$

Name-permutation expressions

Expression $(a\ a') * e$ denotes the result of swapping names a and a' in the structure denoted by the expression e.

Gives rise to a form of non-binding abstraction

$$\mathtt{L}(a, e) \triangleq \mathtt{L}\, a'.\,(a'\ a) * e \qquad [a' \# a, e]$$

a is free in this expression

Name-permutation expressions

Expression $(a\ a')*e$ denotes the result of swapping names a and a' in the structure denoted by the expression e.

Gives rise to a form of non-binding abstraction

$$\mathtt{L}(a,e) \triangleq \mathtt{L}\, a'.\, (a'\ a)*e \qquad [a' \# a, e]$$

Makes NST's form of primitive recursion sufficiently expressive.

$$\begin{array}{rcll} f\,(\mathtt{V}\,a) & = & f_1\,a & \\ f\,(\mathtt{A}\,t\,t') & = & f_2\,t\,t'\,(f\,t)\,(f\,t') & \\ f\,(\mathtt{L}\,a.\,t) & = & \nu a.\, f_3\,a\,t\,(f\,t) & \quad a \# f_1, f_2, f_3 \end{array}$$

e.g. for capture-avoiding substitution $f_3\,a\,t\,x \triangleq \mathtt{L}(a,x)$

Main results

A new model of Odersky's $\nu a.(-)$
using Gabbay-Pitts nominal sets

$\Downarrow$

Decidability of the NST conversion relation
proved by a normalization-by-evaluation argument

$\Downarrow$

All α-structurally recursive functions [JACM 53(2006)]
can be adequately represented in Nominal System T.

Future work

- **Dependent Types**
  Extend NST's primitive recursion to encompass structural induction mod α in versions of

  Agda [Martin-Löf TT]

  Coq [Calculus of Inductive Constructions]

  with Odersky-ν + name-permutations.

Future work

- **Dependent Types**
  Extend NST's primitive recursion to encompass structural induction mod α in versions of

  Agda [Martin-Löf TT]

  Coq [Calculus of Inductive Constructions]

  with Odersky-ν + name-permutations.
- **(Pure) Functional Programming**
  Operational semantics: Odersky-ν+name-permutations

  \+ call-by-value/name functions. . . OK
  \+ call-by-need functions. . . ???

  Relationship of Odersky-ν with dynamic allocation?