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 $\alpha$-equivalence via Odersky-style local names + name-permutations.

NST formalizes common practice with bound names, e.g. . .
\( \lambda \)-terms \( t = \lambda \)-trees mod \( \alpha \)-equivalence

\[
\begin{align*}
a & \mapsto \forall a & \text{variables} \\
t, t' & \mapsto A t t' & \text{application terms} \\
a, t & \mapsto \lambda a. t & \lambda \text{-abstraction terms}
\end{align*}
\]
Typical e.g. of “not quite” primitive recursion:

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

(capture-avoiding substitution)

is well-(and totally-)defined by:

\[
\begin{align*}
    f (V a) &= \text{if } a = a_1 \text{ then } t_1 \text{ else } V a \\
    f (A t t') &= A (f t) (f t') \\
    f (L a. t) &= L a. (f t) \text{ if } a \neq a_1, t_1
\end{align*}
\]
In general:

\[
\begin{align*}
    f(Va) &= f_1 a \\
    f(At t') &= f_2 tt' (ft) (ft') \\
    f(La.t) &= f_3 at (ft) \quad \text{if } a \not\equiv f_1, f_2, f_2
\end{align*}
\]
In general:

\[
\begin{align*}
  f(Va) &= f_1 a \\
  f(At t') &= f_2 t t' (f t) (f t') \\
  f(La.t) &= f_3 a t (f t) \\
\end{align*}
\]

if \( a \neq f_1, f_2, f_3 \)

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

\[
\begin{align*}
    f(Va) &= f_1 a \\
    f(\lambda t t') &= f_2 t t' (f t) (f t') \\
    f(\lambda a. t) &= \nu a. f_3 a t (f t) \quad [a \# f_1, f_2, f_2] \\
\end{align*}
\]

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

A: use a local scoping construct \( \nu a. (\_\_\_) \) for names
In general:

\[
\begin{align*}
    f(Va) &= f_1a \\
    f(Att') &= f_2tt'(ft)(ft') \\
    f(La.t) &= va.f_3at(ft) [ a \neq f_1, f_2, f_2 ]
\end{align*}
\]

A: use a local scoping construct \textit{va. (—)} for names

which one?!
Dynamic allocation

- Familiar—widely used in practice.
- Stateful: $\text{va.e}$ means “add a fresh name $a'$ to the current state and return $e[a'/a]$”
Dynamic allocation

- Familiar—widely used in practice.
- Stateful: \( va.e \) 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:

<table>
<thead>
<tr>
<th></th>
<th>( \text{fst}(va\ a'.(a,a')) \approx \text{fst}(va.(a,a)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>( \text{snd}(va\ a'.(a,a')) \approx \text{snd}(va.(a,a)) )</td>
</tr>
<tr>
<td>but</td>
<td>( va\ a'.(a,a') \not\approx va.(a,a) )</td>
</tr>
<tr>
<td>(consider case</td>
<td>( [-] ) of ( (x,x') \to x = x' )</td>
</tr>
</tbody>
</table>

(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

\[
\begin{align*}
\nu a. (\lambda x. e) & \cong \lambda x. (\nu a. e) \\
\nu a. (e, e') & \cong (\nu a. e, \nu a. e')
\end{align*}
\]

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

\[
\begin{align*}
\nu a a'. (a, a') & \cong (\nu a a'. a, \nu a a'. a') \\
& \cong (\nu a. a, \nu a. a) \\
& \cong \nu a. (a, a)
\end{align*}
\]
Expression \((a \ a') \ast e\) 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

\[
\text{L}(a, e) \triangleq \text{L} \ a'. \ (a' \ a) \ast e \quad [a' \not\# \ a, e]
\]

\(a\) is free in this expression
Name-permutation expressions

Expression \((a \ a') \ast e\) 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

\[
L(a, e) \triangleq L(a').(a' \ a) \ast e \quad [a' \neq a, e]
\]

Makes NST’s form of primitive recursion sufficiently expressive.

\[
\begin{align*}
f(V \ a) & = f_1 a \\
f(A \ t \ t') & = f_2 t t' (f \ t) (f \ t') \\
f(L \ a. \ t) & = \nu a. f_3 a t (f \ t) \quad a \neq f_1, f_2, f_3
\end{align*}
\]

E.g. for capture-avoiding substitution \(f_3 a t x \triangleq L(a, x)\)
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 \( \alpha \)-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 \( \alpha \) in versions of

  Agda [Martin-Löf TT]

  Coq [Calculus of Inductive Constructions]

  with Odersky-\( \nu \) + 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?