The Suspension Notation as a Calculus of Explicit Substitutions

Gopalan Nadathur
Digital Technology Center and
Department of Computer Science
University of Minnesota

[Joint work with D.S. Wilson and A. Gacek]

The Context of Interest

A representation for lambda terms is desired when these are used as *data structures*.

> Abstractions can be used to capture binding structure as is present in formulas, proofs, programs, types, etc.

Comparing lambda terms modulo lambda conversion rules is important in this context.

This issue has to be dealt with in metalanguage, logical framework and proof assistant implementations.

Requirements of Lambda Term Representation

Two specific constraints are usually important:

- Identity modulo renaming should be easy to determine.

  (Typically means that names should be eliminated)

- Should be possible to look underneath abstractions.

This combination makes the situation challenging: explicit substitution treatments are not very difficult

1. when names are used, or
2. without the need to look inside abstractions.

Why Explicit Substitutions At All?

- Laziness may allow substitutions to be avoided altogether:

  We can determine incompatibility of the terms
  $((\lambda x \lambda y \lambda z((x\ z)\ t))\ (\lambda w\, w))$
  and
  $((\lambda x \lambda y \lambda z((x\ y)\ s))\ (\lambda w\, w))$
  without calculating substitutions on t or s.

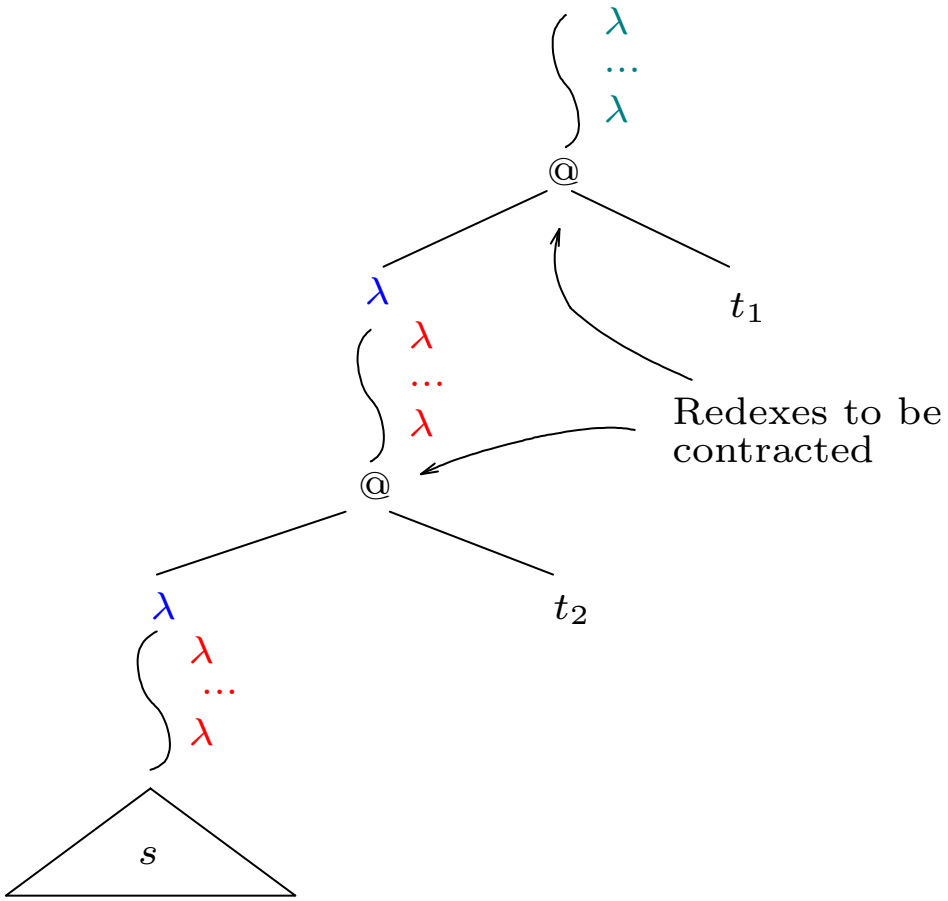
- These are the basis for combining substitution walks:

  In reducing the term
  $((\lambda x \lambda y\, t_1)\ t_2\ t_3)$
  t_2 and t_3 should be substituted simultaneously into t_1.

- In the de Bruijn situation, we also would also like to combine renumbering with other substitutions.

How Do We Make Substitutions Explicit?

The typical scenario:



We want to encode the effect on s of contracting shown redexes.

Encoding Substitutions

Contraction starts at a redex and descends through some abstractions.

Let

ol be the number of abstractions encountered,

nl be the number of abstractions that persist, and

the *nl* value at an intervening abstraction be its *index*.

Then

- *ol* and *nl* give renumbering for external abstractions,
- *nl* and index for persisting abstraction give renumbering for the variable it binds, and
- redex argument and index for contracted abstraction determine the substitution for variable bound by it.

The Simple Suspension Notation

The notation has three categories of expressions called *terms*, *environments* and *environment terms*:

$$\begin{array}{lcl} t & ::= & c \mid x \mid \#i \mid (t\ t) \mid (\lambda\, t) \mid [\![t, n, n, e]\!] \\ e & ::= & nil \mid et :: e \\ et & ::= & (t, n) \end{array}$$

Here x represents variables, c constants, i positive numbers and n natural numbers.

The expression $\#i$ corresponds to de Bruijn indices.

Conceptually, the main change to the syntax is the addition of the expression $[\![t, n, n, e]\!]$ called a *suspension*.

The Rewriting Calculus

Beta Contraction

$$((\lambda t_1)\ t_2) \rightarrow [\![t_1, 1, 0, (t_2, 0) :: nil]\!]$$

The Reading Rules

(r1) $[\![c, ol, nl, e]\!] \rightarrow c \quad (c \text{ is a constant})$

(r2) $[\![x, ol, nl, e]\!] \rightarrow x \quad (x \text{ is a variable})$

(r3) $[\![\#1, ol, nl, (t, l) :: e]\!] \rightarrow [\![t, 0, nl - l, nil]\!]$

(r4) $[\![\#i, 0, nl, e]\!] \rightarrow \#(i + nl)$

(r5) $[\![\#i, ol, nl, et :: e]\!] \rightarrow [\![\#(i-1), (ol-1), nl, e]\!]$, if $i > 1$

(r6) $[\![(t_1\ t_2), ol, nl, e]\!] \rightarrow ([\![t_1, ol, nl, e]\!]\ [\![t_2, ol, nl, e]\!])$

(r7) $[\![\lambda\, t, ol, nl, e]\!] \rightarrow \lambda [\![t, ol + 1, nl + 1, (\#1, nl + 1) :: e]\!]$

Properties of the Simple Calculus

The calculus has several pleasing properties. In particular, let

- $\triangleright_r$ denote the compatible extension of the reading rules, and
- $\triangleright_{\beta_s}$ denote the compatible extension of all the rules.

Then, we have:

Prop 1. $\triangleright_r$ is terminating and confluent.

Prop 2. $\triangleright^*_{\beta_s}$ is capable of simulating beta contraction in the de Bruijn notation.

Prop 3 $\triangleright_{\beta_s}$ is confluent.

Proof Sketch: $\triangleright_{\beta_s}$ rule applications map onto the usual reductions on de Bruijn terms and can also mimic them.

Relationship to Other Calculi

Comparison is meaningful only with calculi without substitution composition.

$\lambda\upsilon$-calculus (Lescanne and colleagues)

- Mapping into suspension terms exists that preserves reductions.
- Single steps are not preserved; $\lambda\upsilon$ is inefficient in renumbering.
- Suspension terms are more general than $\lambda\upsilon$ terms.

λs-calculus (Kamareddine and Rios)

- Mapping into suspension terms exists that preserves reduction.
- λs-calculus separates every substitution and hence has a more efficient "lookup."
- Conversely λs syntax cannot support multiple substitutions.

Meta Variables in Term Syntax

Metavariables are currently required to obey scope rules.

An alternative interpretation that does not require this is obtained by dropping the rule

$$\text{(r2)}\ [\![x, ol, nl, e]\!] \to x \quad (x \text{ is a variable})$$

Unfortunately, the simple suspension calculus is not confluent without this rule. For example the term

$$(\lambda((\lambda\, x)\ t_1))\ t_2$$

has two "normal" forms.

One way to regain confluence is via (directed) substitution permutation rules.

This approach is used in the λs_e and in the λ_{ws} calculus of Guillaume and David.

Composing Substitutions

We desire a rule of the form

$$[\![[\![t_1, ol_1, nl_1, e_1]\!], ol_2, nl_2, e_2]\!] \to [\![t, ol', nl, e']\!]$$

Two reasons for wanting such a rule:

- Regaining confluence after permitting graftable meta variables
- Providing a basis for combining reduction substitutions

In reducing the term

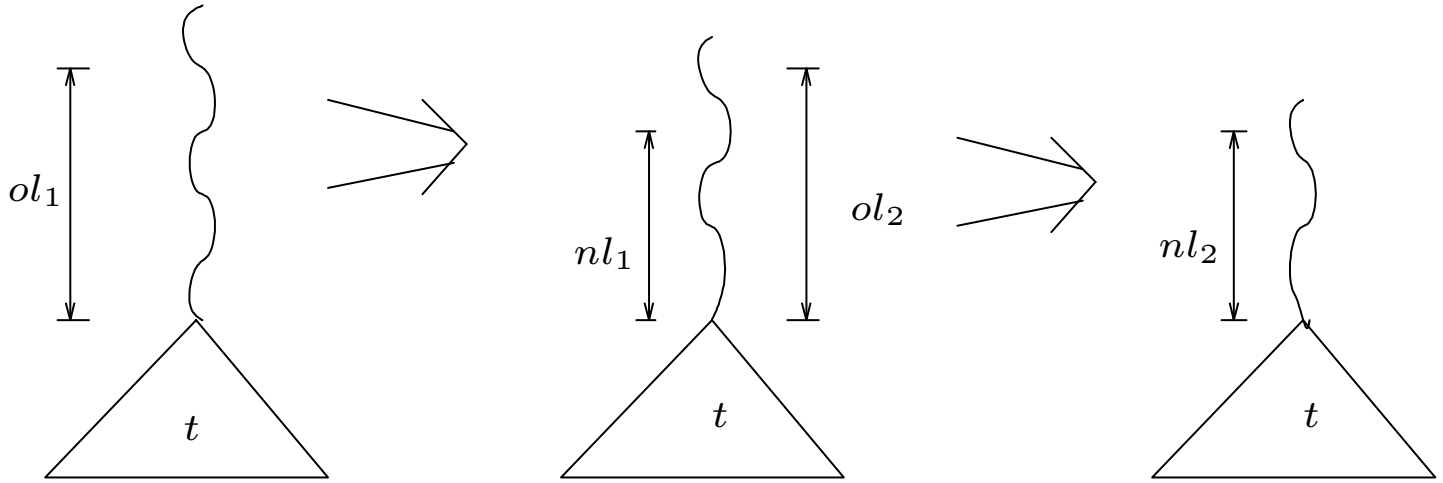
$$((\lambda\, \lambda\, t_1)\ t_2\ t_3)$$

we currently have to perform two walks over t_1

The second reason is a dominating one: combining substitution walks has been shown to be *really* important in practice!

Constructing a Composition Rule

Suppose that ol_2 is larger than or equal to nl_1. The picture is then the following:

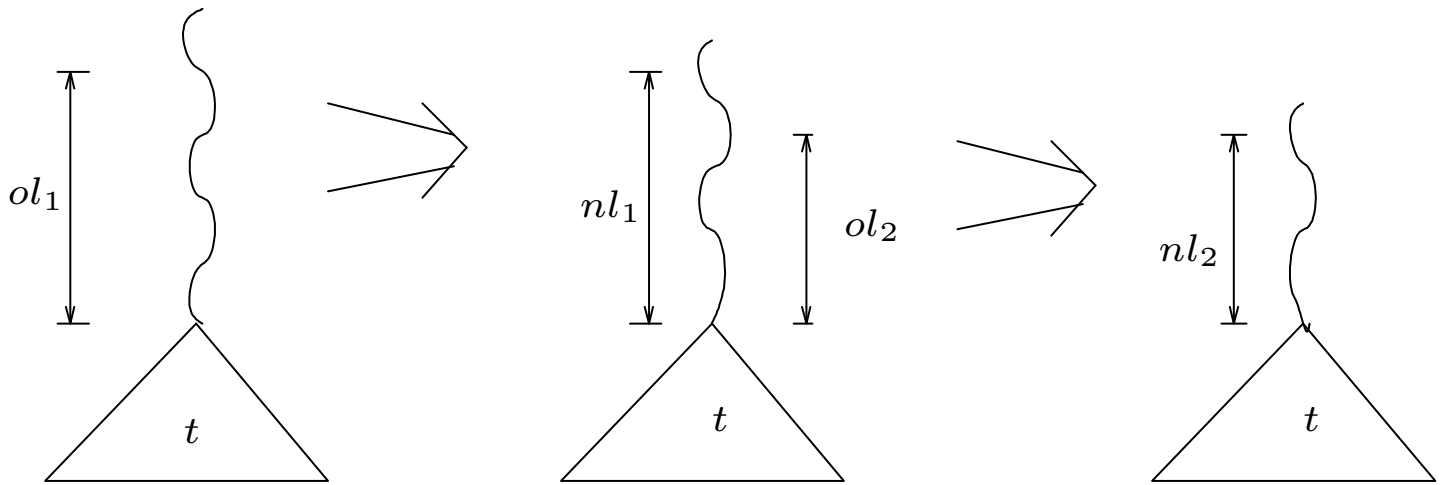


In this case,

- $ol' = ol_1 + (ol_2 - nl_1)$ and $nl' = nl_2$.
- Environment will be e_1 modified by e_2 plus an initial segment of e_2

Constructing a Composition Rule (Contd)

On the other hand, suppose that ol_2 is smaller than nl_1. In this case we have



Now,

- $ol' = ol_1$ and $nl' = nl_2 + (nl_1 - ol_2)$.
- Environment will be e_1, with a final segment of it affected by e_2.

Suspension Notation with Composition

The syntax is enhanced to support the incremental computation of a composed environment:

$$
\begin{array}{lcl}
t & ::= & c \mid x \mid \#i \mid (t\ t) \mid (\lambda\, t) \mid [\![t, n, n, e]\!] \\
e & ::= & nil \mid et :: e \mid \{\!\{e, n, n, e\}\!\} \\
et & ::= & (t, n)
\end{array}
$$

The new environment form $\{\!\{e_1, nl_1, ol_2, e_2\}\!\}$ represents the result of composing e_2 with e_1.

This composition must eventually evaluate to

- elements of (an initial segment of) e_1 modified by e_2, and
- a final segment of either e_1 or e_2.

Substitution Combination Rules

(m1) $[\![[\![t, ol_1, nl_1, e_1]\!], ol_2, nl_2, e_2]\!] \rightarrow [\![t, ol', nl', \{\!\{e_1, nl_1, ol_2, e_2\}\!\}]\!]$,
where $ol' = ol_1 + (ol_2 \dot{-} nl_1)$ and $nl' = nl_2 + (nl_1 \dot{-} ol_2)$

(m2) $\{\!\{e_1, nl_1, 0, nil\}\!\} \rightarrow e_1$

(m3) $\{\!\{nil, 0, ol_2, e_2\}\!\} \rightarrow e_2$

(m4) $\{\!\{nil, nl_1 + 1, ol_2 + 1, et :: e_2\}\!\} \rightarrow \{\!\{nil, nl_1, ol_2, e_2\}\!\}$

(m5) $\{\!\{(t, n) :: e_1, nl_1, ol_2, et :: e_2\}\!\} \rightarrow$
$\{\!\{(t, n) :: e_1, nl_1 - 1, ol_2 - 1, e_2\}\!\}$
provided $nl_1 > n$

(m6) $\{\!\{(t, n) :: e_1, n, ol_2, (s, l) :: e_2\}\!\} \rightarrow$
$([\![t, ol_2, l, (s, l) :: e_2]\!], m) :: \{\!\{e_1, n, ol_2, (s, l) :: e_2\}\!\}$
where $m = l + (n \dot{-} ol_2)$

Properties of the Enhanced Calculus

Let

- $\triangleright_{rm}$ denote the compatible extension of reading and composition rules, and
- $\triangleright_{\beta_s}$ denote the compatible extension of the entire ensemble.

Then, we have:

Prop 1. $\triangleright_{rm}$ is terminating.

(Nontrivial because of (m1), but still true.)

Prop 2. $\triangleright_{rm}$ is locally confluent with or without rule (r2).

(Main complexity is overlap of (m1) with itself. Tedious, but true.)

Prop 3 $\triangleright^*_{\beta_s}$ is capable of simulating beta contraction on de Bruijn terms and is also confluent.

Relationship to Other Calculi

Only other calculi with substitution combination possibilities are the $\lambda\sigma$-calculus and the ΛCCL calculus of (Field'90).

These calculi are virtually identical so we compare only to $\lambda\sigma$.

Proposition. There is a natural mapping from the terms of the suspension calculus to $\lambda\sigma$ terms that preserves reducibility.

Comments

- Recent simplification to the suspension notation facilitates the construction of this mapping.
- Translation is only of terms; environments are not exactly like the "free-floating" substitutions of the $\lambda\sigma$-calculus.

Preservation of Strong Normalizability?

There are simply typeable lambda terms with nonterminating reduction sequences in the $\lambda\sigma$-calculus.

What about the suspension calculus?

The answer is not presently known.

However,

- The known counter-examples for the $\lambda\sigma$-calculus is not one for the suspension notation.
- The interactions between environments seems to be a bit more restricted so as to give some hope in the suspension calculus.

This is, thus, still an open question that is interesting to settle at this point.

Derived Combination Rules

Some of the power of the combination calculus can be abstracted into the following derived rules:

$$(\lambda\,[\![t_1, ol+1, nl+1, (\#1, nl+1) :: e]\!])\ t_2 \rightarrow [\![t_1, ol+1, nl, (t_2, nl) :: e]\!]$$

$$[\![[\![t, ol, nl, e]\!], 0, nl', nil]\!] \rightarrow [\![t, ol, nl+nl', e]\!]$$

The first rule combines reduction substitutions, the second folds renumbering into other substitution walks.

Conjecture: A leftmost outermost reduction procedure that uses these rules will not create nested suspensions.

Note that, with only the derived rules,

- the syntax of the simple calculus is preserved, but
- capabilities for combining substitution walks are obtained.

Annotations on Terms

Terms can be marked to indicate whether or not they are closed.

With such annotations, substitutions can sometimes become trivial.

For example, if c is an annotation for closedness, we have

$$[\![t_c, ol, nl, e]\!] \rightarrow t_c$$

Of course, rules must be modified to preserve annotation information as much as is possible.

Detailed presentation of the calculus appears in [Nadathur, 1999].

It is the derived calculus with annotations that is used both in the *FLINT* system and in the *Teyjus* λProlog implementation.

Conclusion

- The suspension notation is an interesting and important explicit substitution calculus.
- In complementary empirical studies with Liang and Qi, we have tried to quantify the practical utility of its varied features.

  For example, we know that
  - combining ability (via derived rules) and annotations have real practical benefits, and
  - Varied reduction procedures are possible and some unexpected combinations work best in the applications.
- There is at least one important question still to be settled.
- Unification modulo the suspension calculus?

Translation to the $\lambda\sigma$-calculus

- The translation is a direct one for terms and consists of the following:

$$T(\#i) = 1[\uparrow^{(i-1)}]$$
$$T((t_1\ t_2)) = (T(t_1)\ T(t_2)) \qquad T(\lambda\, t) = \lambda\, T(t)$$
$$T(\{\!\{t, nl, ol, e\}\!\}) = T(t)[E(e, nl)]$$

- For environments, we map relative to an index:

$$E(nil, j) = \uparrow^j \qquad E(et :: e, j) = ET(et, j)\,.\,E(e, j)$$
$$E(\{\!\{e_1, nl, ol, e_2\}\!\}, j) = E(e_1, nl) \circ E(e_2, j - (nl \dot{-} ol))$$

- The index component carries over to environment terms:

$$ET((t, i), j) = \uparrow^{(j-i)}$$

Of course, the index must be "sensible" for the mapping to work.

However, wellformedness conditions on suspension expressions guarantee this.