Electronic Notes in Theoretical Computer Science 67 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume67.html 13 pages

The Suspension Notation for Lambda Terms and its Use in Metalanguage Implementations

Gopalan Nadathur [1,2]

Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455, U.S.A

Abstract

Many metalanguages and logical frameworks have emerged in recent years that use the terms of the lambda calculus as data structures. A common set of questions govern the suitability of a representation for lambda terms in the implementation of such systems: α-convertibility must be easily recognizable, sharing in reduction steps, term traversal and term structure must be possible, comparison and unification operations should be efficiently supported and it should be possible to examine terms embedded inside abstractions. Explicit substitution notations for lambda calculi provide a basis for realizing such requirements. We discuss here the issues related to using one such notation—the suspension notation of Nadathur and Wilson—in this capacity. This notation has been used in two significant practical systems: the Standard ML of New Jersey compiler and the Teyjus implementation of λProlog. We expose the theoretical properties of this notation, highlight pragmatic considerations in its use in implementing operations such as reduction and unification and discuss its relationship to other explicit substitution notations.

1 Introduction

Metalanguages and logical frameworks manipulate a variety of symbolic objects such as formulas, programs, proofs and types whose structures naturally involve the notion of binding. Lambda terms have been found to be a useful in capturing the abstract syntax of such objects. Suppose, for example, that we wish to represent the formula $\forall x((p\ x) \vee (q\ c))$ in which p and q are predicate names and c is a constant. Noting that a quantifier plays the dual role of determining a scope and of making a predication, the essential structure of this formula can be captured by the lambda term $(all\ (\lambda x\, or((p\ x), (q\ c))))$; in this

[1] This work has been partially supported by the NSF under the grant CCR-0096322.
[2] Email: gopalan@cs.umn.edu

term, *all* is a constructor that represents universal quantification and *or* is a constructor that represents disjunction. The explicit treatment of binding in this representation makes for a simple and transparently correct implementation of several logical operations on formulas. Thus, the task of instantiating with t the quantifier in a formula represented by the term $(all\ P)$ is realized immediately by writing the term $(P\ t)$. Actual substitution is carried out, with all the necessary renamings, by the β-reduction operation on lambda terms. Similarly, structure analysis of formulas that is sensitive to binding can be performed through an enhanced unification operation. For example, suppose that we wish to recognize that the given formula is one that has a universal quantification over a disjunction where the quantified variable does not appear in the second disjunct. This property can be ascertained by attempting to unify the term representing it with the 'template' $(all\ (\lambda x\, or((P\ x), Q)))$ in which P and Q are instantiatable variables. The variable Q here cannot be substituted for in such a way that the second disjunct comes to depend on the quantifier and will therefore only match with the 'right' kind of term.

The programming convenience of such higher-order abstract syntax must, of course, be complemented by an efficient representation for lambda terms within the implementation of the relevant metalanguage or logical framework. While lambda term realizations have long been of interest in the functional programming context, the present *intensional* use of these terms places new constraints on adequate representations. Thus, the comparison of lambda terms must be possible and so their structures cannot be sacrificed in a compilation process. At a more detailed level, the notion of equality between lambda terms must ignore the particular names used for bound variables. For this reason, the representation must support the rapid determination of identity up to α-convertibility. Another operation that is important to realize efficiently is β-reduction. For reasons that we discuss later, two requirements must be satisfied relative to this operation: it should be possible to perform the substitutions generated by β-contractions *lazily* and to percolate such substitutions as well as to perform β-contractions *inside* abstraction contexts. Finally, the higher-order unification computation is central to many metaprogramming tasks and consideration must be given to the treatment of meta variables and to operations that are important in its implementation.

A good starting point for an adequate intensional representation of lambda terms is the de Bruijn notation for lambda terms [3]. This notation eliminates names for bound variables, thus simplifying identity checking modulo renaming. Explicit substitution notations [1,2,4,8,13] that build on the de Bruijn scheme provide the basis for meeting several of the other mentioned requirements. There are differences in the specific characteristics of such notations and choices must also be made in the specific manner in which these are to be deployed in the context of metalanguage implementation. This paper exposes some of the issues that are important in this situation, gleaned from our experience in realizing the language λProlog. We orient the discussion around the suspension notation of Nadathur and Wilson that, to our knowledge, is the only one to be used in two actual implementation tasks [12,14]. However, our general comments apply to other schemes as well and we also compare the different notations at the end.

2 The Suspension Notation

The combination of substitution walks that arise from contracting different β-redexes can have a significant impact on efficiency. Thus, suppose that we wish to instantiate the two quantifiers in the formula represented by $(all\ (\lambda x\,(all\ (\lambda y\,P))))$, where P represents an unspecified formula, with the terms t_1 and t_2. Assuming a de Bruijn representation, such an instantiation is realized through two contractions, eventually requiring t_2 and t_1 to be substituted for the first and second free variables in P and the indices of all other free variables to be decremented by two. Each of these substitutions involves a walk over the *same* structure—the structure of P—and it would be profitable if they could all be done together. Studies reveal that, by systematically exploiting this idea, structure traversal can be substantially reduced in practice, down to as little as an eighth of the original in some cases [9]. Now, an ability that is critical to combining walks in this manner is that of temporarily suspending substitutions generated by β-contractions. In a situation in which all the redexes are available in a single term, this kind of delaying of substitution can be built into the reduction procedure through *ad hoc* devices. However, in the case being considered, the two quantifier instantiations are ones that can only be considered incrementally and, further, intervening structure needs to be processed before the abstraction giving rise to the second redex is encountered. The structure that leads to sharing is therefore not all available within a single call to a reduction procedure and an explicit encoding of substitution over P seems to be necessary for realizing this benefit.

Substitutions are delayed in the implementation of functional programming languages by using environments and it may appear that a simple reflection of such environments into term structure should suffice for the present purposes. The problem, however, is that when lambda terms are used to represent objects, it may be necessary to examine structure embedded *inside* abstractions. Consider, for example, the task of determining if the term that results from instantiating the quantifier in a formula of the form $(all\ R)$ has a shape that is captured by the template $(all\ (\lambda x\, or((P\ x), Q)))$; we assume that R represents an unspecified term here and that P and Q are instantiatable variables. A positive determination involves percolating a substitution underneath the abstraction corresponding to a quantifier and then checking if the embedded structure is a disjunction. In carrying out this computation it is necessary to consider α-conversion or an equivalent renumbering in the de Bruijn representation, something whose incorporation into a environment model requires care. Notice also that the actual form of R may require β-contractions to be performed within the abstraction capturing the quantifier scope in order to reveal its top-level logical structure. This kind of calculation, further complicates the structure of environments.

The suspension notation embodies a solution to the problems described above. Formally, it encompasses a collection of expressions called terms, environments and environment terms whose syntax is given by the categories $\langle T\rangle$, $\langle E\rangle$ and $\langle ET\rangle$ defined by the following grammar rules in which $\langle C\rangle$, $\langle I\rangle$ and $\langle N\rangle$ represent constants, positive numbers and natural numbers, respectively:

$$\begin{array}{lll}\langle T\rangle & ::= & \langle C\rangle \mid \#\langle I\rangle \mid (\langle T\rangle\ \langle T\rangle) \mid (\lambda\,\langle T\rangle) \mid [\![\langle T\rangle,\langle N\rangle,\langle N\rangle,\langle E\rangle]\!] \\ \langle E\rangle & ::= & nil \mid \langle ET\rangle :: \langle E\rangle \mid \{\!\{\langle E\rangle,\langle N\rangle,\langle N\rangle,\langle E\rangle\}\!\} \\ \langle ET\rangle & ::= & @\langle N\rangle \mid (\langle T\rangle,\langle N\rangle) \mid \langle\!\langle\langle ET\rangle,\langle N\rangle,\langle N\rangle,\langle E\rangle\rangle\!\rangle.\end{array}$$

The essential addition to de Bruijn terms to produce suspension terms is that of expressions of the form $[\![t, ol, nl, e]\!]$, where t is a term and e is an environment. Such a term, referred to as a suspension, represents the term t with its first ol variables substituted for in a way determined by the environment e and its remaining bound variables renumbered to reflect the fact that t used to appear within ol abstractions but now appears within nl of them. In the simplest form, the elements of an environment are either substitution terms generated by contractions or are dummy entries representing abstractions that persist in an outer context. However, renumbering of indices may have to be done during substitution, and, to encode this, each such environment element is annotated by a relevant abstraction level referred to as its index. Such suspensions must satisfy certain wellformedness constraints that have a natural basis in our informal understanding of their content: in an expression of the form $[\![t, i, j, e]\!]$, the 'length' of the environment e must be equal to i, the indices of the entries in e must be non-increasing and they must be bounded by j. The notation also allows for the combination of substitutions: an expression of the form $\{\!\{e_1, i, j, e_2\}\!\}$ represents the composition of the substitutions contained in e_1 and e_2 and $\langle\!\langle et, i, j, e_2\rangle\!\rangle$ corresponds to the environment term et modified by the substitutions in the environment e_2. The numbers i, j, the lengths of environments and the indices of terms in the environments being composed must satisfy certain constraints that arise naturally out of the restrictions discussed on simple environments. Space limitations prevent a discussion of these aspects here, but a detailed treatment appears in [13].

The usual β-contraction operation is realized in the suspension notation in two phases: the generation and the subsequent percolation of a substitution. This process is described formally by a collection of rewrite rules. These rules are broken up into three categories: the β_s rule that generates suspensions, the reading rules that percolate substitutions and the merging rules that permit intermediate suspensions to be combined. These rule categories are presented in Figures 1, 2 and 3, respectively. The merging rules are actually redundant from the perspective of simulating β-reduction. However, without them it is not possible to combine substitutions and the walks that effect them.

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

Fig. 1. The β_s rule

(r1) $\quad [\![c, ol, nl, e]\!] \rightarrow c$, provided c is a constant.

(r2) $\quad [\![\#i, 0, nl, nil]\!] \rightarrow \#j$, where $j = i + nl$.

(r3) $\quad [\![\#1, ol, nl, @l :: e]\!] \rightarrow \#j$, where $j = nl - l$.

(r4) $\quad [\![\#1, ol, nl, (t, l) :: e]\!] \rightarrow [\![t, 0, nl', nil]\!]$, where $nl' = nl - l$.

(r5) $\quad [\![\#i, ol, nl, et :: e]\!] \rightarrow [\![\#i', ol', nl, e]\!]$,
where $i' = i - 1$ and $ol' = ol - 1$, provided $i > 1$.

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

(r7) $\quad [\![(\lambda\, t), ol, nl, e]\!] \rightarrow (\lambda\, [\![t, ol', nl', @nl :: e]\!])$,
where $ol' = ol + 1$ and $nl' = nl + 1$.

Fig. 2. The reading rules

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

(m2) $\quad \{\!\{nil, nl, 0, nil\}\!\} \rightarrow nil$.

(m3) $\quad \{\!\{nil, nl, ol, et :: e\}\!\} \rightarrow \{\!\{nil, nl', ol', e\}\!\}$,
where $nl, ol \geq 1$, $nl' = nl - 1$ and $ol' = ol - 1$.

(m4) $\quad \{\!\{nil, 0, ol, e\}\!\} \rightarrow e$.

(m5) $\quad \{\!\{et :: e_1, nl, ol, e_2\}\!\} \rightarrow \langle\!\langle et, nl, ol, e_2\rangle\!\rangle :: \{\!\{e_1, nl, ol, e_2\}\!\}$.

(m6) $\quad \langle\!\langle et, nl, 0, nil\rangle\!\rangle \rightarrow et$.

(m7) $\quad \langle\!\langle @n, nl, ol, @l :: e\rangle\!\rangle \rightarrow @m$,
where $m = l + (nl \dot{-} ol)$, provided $nl = n + 1$.

(m8) $\quad \langle\!\langle @n, nl, ol, (t, l) :: e\rangle\!\rangle \rightarrow (t, m)$,
where $m = l + (nl \dot{-} ol)$, provided $nl = n + 1$.

(m9) $\quad \langle\!\langle (t, nl), nl, ol, et :: e\rangle\!\rangle \rightarrow ([\![t, ol, l', et :: e]\!], m)$
where $l' = ind(et)$ and $m = l' + (nl \dot{-} ol)$.

(m10) $\quad \langle\!\langle et, nl, ol, et' :: e\rangle\!\rangle \rightarrow \langle\!\langle et, nl', ol', e\rangle\!\rangle$,
where $nl' = nl - 1$ and $ol' = ol - 1$, provided $nl \neq ind(et)$.

Fig. 3. The merging rules

Definition 2.1 *The (one-step) reduction relations on suspension expressions generated by the reading and merging rules on one hand and by all the rules on the other are denoted by* $\rhd_{rm}$ *and* $\rhd_{rm\beta_s}$*, respectively. The usual* β*-contraction relation on de Bruijn terms is denoted by* $\rhd_\beta$*. Finally, we denote the reflexive transitive closure of a relation* R *by* R^**.*

The reading and merging rules are intended to expose the de Bruijn term underlying any given term and we would expect them to always succeed in doing so. The following proposition, proved in [13], shows this to be the case.

Proposition 2.2 *The relation $\rhd_{rm}$ is strongly terminating, i.e. all sequences of such reductions are finite.*

A term of the form $[\![[\![[\![t, ol_1, nl_1, e_1]\!], ol_2, nl_2, e_2]\!], ol_3, nl_3, e_3]\!]$ can be 'flattened' into a single suspension by two uses of the rule m1. However, this flattening can be achieved in two different ways—by first composing e_1 and e_2 and then composing the result with e_3 or by first composing e_2 and e_3 and then composing e_1 with the result—and we would like the outcome to be the same in either case. Some explicit substitution calculi guarantee this by including an associativity rule for composing substitutions. In our calculus, this property is a consequence of the other rules:

Proposition 2.3 *Let a and b be environments of the form*

$$\{\!\{\{\!\{e_1, nl_1, ol_2, e_2\}\!\}, nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3\}\!\}$$

and

$$\{\!\{e_1, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \{\!\{e_2, nl_2, ol_3, e_3\}\!\}\}\!\},$$

respectively. Then there is an environment r such that $a \rhd^*_{rm} r$ and $b \rhd^*_{rm} r$.

We would like a stronger property that the existence of $\rhd_{rm}$ normal forms to hold: these forms should be unique for any given expression. In light of Proposition 2.2, it is enough to show that $\rhd_{rm}$ is locally confluent. Towards this end, we need to consider the nontrivial overlaps in the lefthand sides of our rules and to show that the critical pairs corresponding to these can be rewritten to a common form. The relevant overlaps are between m1 and each of the reading rules, m1 and itself and m2 and m4. The only complicated case amongst these is when the overlap is between m1 and itself. However, Proposition 2.3 ensures reducibility to a common form in this case. Thus, we have

Proposition 2.4 *The relation $\rhd_{rm}$ is locally confluent and, hence, confluent.*

We shall depict the $\rhd_{rm}$ normal form of an expression e in the suspension calculus by $|e|$. The correspondence between the reduction relations on de Bruijn terms and suspension terms can then be stated as follows.

Proposition 2.5 *Let t be a term in the suspension calculus. If $t \rhd^*_{rm\beta_s} r$ then $|t| \rhd^*_{\beta} |r|$. Conversely, if $|t| \rhd^*_{\beta} s$, then $t \rhd^*_{rm\beta_s} s$.*

From this proposition it follows also that $\rhd_{rm\beta_s}$ is confluent.

3 Eliminating the Merging Rules

The merging rules provide a versatile mechanism for combining substitutions. However, their power derives from a fine-grained treatment of composition that is a little cumbersome for actual implementation. For this reason, it is worthwhile to explore the possibility of capturing their common uses in coarser, more efficient, derived rules. We observe two situations below to which this approach can be effectively applied.

The first situation corresponds to the combination of substitutions arising from the contraction of nested β-redexes. As an illustration, we might consider the reduction of the term $((\lambda\,((\lambda\,(\lambda\,((\#1\ \#2)\ \#3)))\ t_2))\ t_3)$, in which t_2 and t_3 are arbitrary de Bruijn terms. In a leftmost-outermost reduction regime, the first step would be to use the β_s rule to produce the suspension

$$[\![((\lambda\,(\lambda\,((\#1\ \#2)\ \#3)))\ t_2), 1, 0, (t_3, 0) :: nil]\!].$$

The reading rules would now be used a few times to produce the term

$$((\lambda\,[\![(\lambda\,((\#1\ \#2)\ \#3)), 2, 1, @0 :: (t_3, 0) :: nil]\!])\ [\![t_2, 1, 0, (t_3, 0) :: nil]\!]).$$

At this stage, the β_s rule would be used again, yielding the term

$$[\![[\![(\lambda\,((\#1\ \#2)\ \#3)), 2, 1, @0 :: (t_3, 0) :: nil]\!], 1, 0, \\ ([\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0) :: nil]\!].$$

The m1 rule can now be used to compose the two environments, yielding

$$[\![(\lambda\,((\#1\ \#2)\ \#3)), 2, 0, \\ \{\!\{@0 :: (t_3, 0) :: nil, 1, 1, ([\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0) :: nil\}\!\}]\!].$$

Using the other merging rules, this term can be reduced to the form

$$[\![(\lambda\,((\#1\ \#2)\ \#3)), 2, 0, ([\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0) :: (t_3, 0) :: nil]\!]$$

whose virtue is that 'lookups' of its environment are simple.

The sequence of rewriting steps starting from the second use of the β_s rule and ending in the final suspension term can be collapsed into one use of a more 'powerful' β_s rule: [3]

$$(\beta_s') \qquad ((\lambda\,[\![t_1, ol+1, nl+1, @nl :: e]\!])\ t_2) \rightarrow [\![t_1, ol+1, nl, (t_2, nl) :: e]\!]$$

This rule can be shown to be a derived rule of the suspension calculus. The advantage to using it is that the intermediate merging steps can be avoided.

The example just considered actually illuminates a tradeoff between sharing in structure walks realized through merging and sharing in reduction. After the first use of the β_s rule, we chose above to propagate substitutions. We could have chosen to rewrite the inner β_s-redex instead, producing

$$[\![[\![(\lambda\,((\#1\ \#2)\ \#3)), 1, 0, (t_2, 0) :: nil]\!], 1, 0, (t_3, 0) :: nil]\!].$$

In a graph-based implementation of reduction, following this course ensures that this rewriting step is carried out before substitution propagation breaks any sharing relative to it. Note that to fully realize the benefits of such sharing, it is necessary to perform the two substitutions embedded in the term in *separate* walks over the structure of $\lambda\,((\#1\ \#2)\ \#3)$. There is, thus,

[3] There is an unstated proviso on this rule that holds of all terms derivable from de Bruijn ones using our reduction rules: the index of terms in e must be less than that of $@nl$.

a dilemma between two different choices in reduction. However, this dilemma is genuine only when there are real cases of shared redexes. Our experiments reveal very few such situations in practice [9], indicating a preference for an approach that attempts to combine structure traversals.

The second situation in which merging rules are useful arises when indices need to be renumbered in a suspension that is substituted inside an abstraction context. We illustrate this by continuing the reduction of the term $((\lambda\,((\lambda\,(\lambda\,((\#1\ \#2)\ \#3)))\ t_2))\ t_3)$. Using the reading rules from where we left off, this term can be transformed into

$$\begin{array}{l}(\lambda\,((\#1\ [\![[\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0, 1, nil]\!]) \\ \qquad\qquad [\![\#3, 3, 1, @0 :: ([\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0) :: (t_3, 0) :: nil]\!])).\end{array}$$

The subterm $[\![[\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0, 1, nil]\!]$ here corresponds to t_2 embedded within two suspensions, with the outer suspension representing a 'bumping up' of the indices for the free variables in the inner suspension, necessitated by its insertion inside an abstraction. Using the merging rules, the indicated subterm can be rewritten into $[\![t_2, 1, 1, (t_3, 0) :: nil]\!]$, thereby combining the different substitutions into one environment.

This use of the merging rules can also be reflected into a derived rule:

$$\text{(bump)} \qquad [\![[\![t, ol, nl, e]\!], 0, nl', nil]\!] \rightarrow [\![t, ol, nl + nl', e]\!].$$

In an actual implementation of reduction, this rule can, in fact, be rolled into the application of the reading rule r4.

The disadvantage of the bump rule is that it, once again, prefers sharing in structure traversal to sharing in reduction. Actual loss in reduction sharing here is also something that differentiates between the de Bruijn representation and a name based representation of bound variables in implementing β-reduction: using the bump rule, the extra renumbering work in the de Bruijn scheme is subsumed into an already necessary traversal of the structure of the embedded term but with a possible loss in reduction sharing. As before, our observation has been that in practice there are very few real opportunities for sharing in reduction, indicating a preference for the bump rule whenever it is applicable and also little downside in reduction to using the de Bruijn scheme.

Definition 3.1 *We denote the reduction relation defined by the reading and the bump rules by* $\rhd_{r'}$. *The relation obtained when the* β_s *and the* β'_s *rules are also included is denoted by* $\rhd_{r\beta'_s}$.

The following proposition shows the coherence of our derived rules.

Proposition 3.2 *The* $\rhd_{r'}$ *relation is confluent and strongly terminating. Further, for any term* t, *if* $t \rhd^*_{r\beta'_s} r$ *then* $t \rhd^*_{rm\beta_s} r$. *Conversely, if* $t \rhd^*_{rm\beta_s} r$, *then there are terms* s *and* s' *such that* $t \rhd^*_{r\beta'_s} s$, $r \rhd^*_{rm} s'$ *and* $s \rhd^*_{rm} s'$. *Finally* $\rhd_{r\beta'_s}$ *is confluent.*

The reduction of a de Bruijn term to (head) normal form may be carried out using solely the rules defining the $\rhd_{r\beta'_s}$ relation. The main disadvantage to not using the merging rules is that some opportunities for sharing in structure walks may be missed. It turns out that, with an leftmost-outermost implementation of reduction, there are very few such cases in practice.[4]

4 Instantiatable Variables, Confluence and Unification

The current syntax of suspension expressions does not allow for instantiatable or meta variables. Such variables may be introduced in one of two forms.

In the first form, these variables would be treated just as in the normal lambda calculus. In particular, instantiations for them must respect the notion of scope. Thus, if X is an instantiatable variable occurring within abstractions binding $x_1, \ldots, x_n$, then it cannot be replaced by a structure that depends on any of the abstractions. This logical view is actually the one that is needed in pattern recognition applications. The term $(all\ (\lambda x\, or((P\ x), Q)))$, for instance, functions as a recognizer for formulas with a universal quantification over a disjunction whose right part is independent of the quantifier precisely because Q cannot be instantiated to a form that depends on x.

Building this view of instantiatable variables into the suspension notation is easy. At the level of syntax, we simply change the rule for terms to

$$\langle T \rangle \quad ::= \quad \langle V \rangle \mid \langle C \rangle \mid \#\langle I \rangle \mid (\langle T \rangle\ \langle T \rangle) \mid (\lambda\, \langle T \rangle) \mid [\![\langle T \rangle, \langle N \rangle, \langle N \rangle, \langle E \rangle]\!]$$

where $\langle V \rangle$ represents the category of such variables. To account for the fact that these variables cannot be affected by substitutions generated by β-contractions, we add the following to our reading rules:

(r8) $[\![x, ol, nl, e]\!] \rightarrow x$, provided x is an instantiatable variable.

This rule is similar to the one for reading constants. Thus, it should not be difficult to see that confluence and termination properties extend naturally to the syntax that includes the new variables. Note also that the smaller collection of rewrite rules discussed in Section 3 suffices for reducing terms containing such variables to normal form.

The other possibility is to view instantiatable variables as placeholders against which any wellformed term can be grafted. Such 'graftable' variables appear initially to fly in the face of pattern matching applications. However, the necessary constraints for such applications can be built in through suitable preprocessing. Thus, consider the template $(all\ (\lambda x\, or((P\ x), Q)))$ that in de Bruijn notation would be written as $(all\ (\lambda\, (or\ (P\ \#1)\ Q)))$. This term may be transformed into $(all\ (\lambda\, (or\ ([\![P, 0, 1, nil]\!],\ \#1)\ [\![Q, 0, 1, nil]\!])))$. By so embedding P and Q inside suspensions, we insulate them from a dependence on the external abstraction.

This kind of a view can also be incorporated into the suspension notation. The syntax of terms needs to be modified exactly as before. In contrast

[4] There should, in fact, be no such cases if reduction is the sole operation on terms. All observed cases originate from unification substitutions for the meta variables discussed later.

to the earlier situation, however, no new rewrite rules need be added. The rationale is that the effect of reduction substitutions on instantiatable variables is unknown until their instantiations are themselves known. At a point where these variables have been instantiated, the rewrite rules pertaining to the other forms for terms suffice for computing the effects of reductions.

Let us denote by $\rhd_{rm}$, $\rhd_{r'}$, $\rhd_{rm\beta_s}$ and $\rhd_{r\beta'_s}$ the previously seen reduction relations on the extended syntax. While $\rhd_{rm}$ and $\rhd_{r'}$ must still be strongly terminating, confluence properties are more problematic. The relation $\rhd_{r\beta'_s}$ is, in fact, not confluent. Thus, consider the term $((\lambda\,((\lambda\,X)\ t_1))\ t_2)$ in which X is an instantiatable variable and t_1 and t_2 are terms in normal form. Three distinct terms may be posited as 'normal' forms for this:

$[\![[\![X, 1, 0, (t_1, 0) :: nil]\!], 1, 0, (t_2, 0) :: nil]\!]$,
$[\![[\![X, 2, 1, @0 :: (t_2, 0) :: nil]\!], 1, 0, ([\![t_1, 1, 0, (t_2, 0) :: nil]\!], 0) :: nil]\!]$, and
$[\![X, 2, 0, ([\![t_1, 1, 0, (t_2, 0) :: nil]\!], 0) :: (t_2, 0) :: nil]\!]$.

Adding the merging rules changes the picture: the first two terms then reduce to the last. The following proposition can, in fact, be shown:

Proposition 4.1 *Assuming a collection of terms that includes graftable variables, the relations $\rhd_{rm}$ and $\rhd_{rm\beta_s}$ are both confluent.*

The key observation in the proof of this proposition is that associativity for composing substitutions as described in Proposition 2.3 continues to hold.

Interest in the graftable interpretation of meta variables arises from the new approach to higher-order unification described in [5] that exploits such variables. The usual procedure [7] for the (typed) lambda calculus is based on reducing any given unification problem into a set of equations of the form

$$\lambda x_1 \ldots \lambda x_n\,(X\ t_1\ \ldots\ t_m) = \lambda x_1 \ldots \lambda x_n\,(@\ s_1\ \ldots\ s_l)$$

where X is an instantiatable variable and $@$ is a constant or one of the variables $x_1, \ldots, x_n$. Towards solving such an equation, substitutions of the form

$$\lambda w_1 \ldots \lambda w_m\,(@'\ (H_1\ w_1\ \ldots\ w_m)\ \ldots\ (H_k\ w_1\ \ldots\ w_m)),$$

where $@'$ is either $@$ or one of $w_1, \ldots, w_m$ and $H_1, \ldots, H_k$ are new instantiatable variables, are posited for X. Such substitutions try to get the heads of the two terms that are to be unified to match while delaying decisions concerning the arguments. The arguments of the substitution term are, in fact, chosen so as to not preclude any dependencies on the arguments of the original term. For example, if $@' = @$ and, correspondingly, $l = k$, then this substitution will reduce the unification problem to one of simultaneously solving the equations

$$\lambda x_1 \ldots \lambda x_n\,(H_i\ t_1\ \ldots\ t_m) = \lambda x_1 \ldots \lambda x_n\, s_i$$

for $1 \le i \le l$. Note that H_i is free to 'use' the arguments $t_1, \ldots, t_m$ in any fashion deemed necessary.

The above transformation involves the construction of a complicated term, the contraction of several β-redexes and a subsequent calculation of their substitution effects. Using explicit substitutions and 'graftable' variables the effort involved in this percolation of dependency information can be considerably reduced. By substituting the term $\lambda w_1 \ldots \lambda w_m\, Y$ where Y is a graftable variable for X, the original equation can be reduced at the outset to

$$\lambda x_1 \ldots \lambda x_n\, [\![Y, m, 0, (t_m, 0) :: \ldots :: (t_1, 0) :: nil]\!] = \lambda x_1 \ldots \lambda x_n\, (@\ s_1\ \ldots\ s_l).$$

Notice that the considered substitution for X is meaningful only if Y can later be replaced with something that might contain the variables $w_1, \ldots, w_m$, *i.e.*, Y *must* be graftable. Now, after this reduction, a term of the form $(@\ H_1\ \ldots\ H_l)$ can be posited for Y, allowing the equation to be transformed into ones of the form

$$\lambda x_1 \ldots \lambda x_n\, [\![H_i, m, 0, (t_m, 0) :: \ldots (t_1) :: nil]\!] = \lambda x_1 \ldots \lambda x_n\, s_i$$

for $1 \leq i \leq l$. Significantly, the formation of a complicated term involving applications and the subsequent reductions simply for the purpose of transmitting dependency information can be avoided.

The above discussion actually indicates a tradeoff between different approaches to implementing higher-order unification. The approach based on graftable variables has the mentioned benefits but it also requires the use of a more complete, and complicated, set of environment merging rules. An interesting observation is that the new approach to unification depends mainly on the generation of (head) normal forms that do not contain nested suspensions at the top level. A possibility is that a special control regimen with a reduced set of rewrite rules will ensure that only such forms are produced.

5 Comparison with Other Explicit Substitution Calculi

Three properties are coveted for explicit substitution notations: confluence in a situation where graftable meta variables are included, the ability to compose substitutions and the preservation of strong normalizability for terms in the underlying lambda calculus. Of these, combinability of substitutions seems to be the most important for metalanguage implementations. Unfortunately, most explicit substitution calculi seem not to include this facility. Particular calculi sacrifice other properties as well. The $\lambda\upsilon$-calculus preserves strong normalizability [2] but it does not admit meta variables. The λs_e-calculus permits meta variables and is confluent even with this addition [8] but does not preserve strong normalizability [6]. The λ_{ws_o}-calculus alone both admits meta variables and preserves strong normalizability [4].

The two calculi that do permit the composition of substitutions are the $\lambda\sigma$-calculus [1] and the suspension notation. There are several similarities between the two calculi that we hope to demonstrate via translation functions in a longer paper. We restrict ourselves here to mentioning two *differences* that might be significant to low-level implementation tasks. First, it appears easier in our calculus to separate out rewrite rules based on function and to thereby identify subsets like that in Section 3 that are easier to use in practice. The second difference concerns the way in which the adjustments to the indices of terms in the environment are encoded. In our notation, these are not maintained explicitly but are obtained from the difference between the embedding level of the term that has to be substituted into and an embedding level recorded with the term in the environment. Thus, consider a suspension term of the form $[\![t_1, 1, nl, (t_2, nl') :: nil]\!]$. This represents a term that is to be obtained by substituting t_2 for the first free variable in t_1 (and modifying the indices for the other free variables). However, the indices for the free variables in t_2 must be 'bumped up' by $(nl - nl')$ before this substitution is made. In the $\lambda\sigma$-calculus, the needed increment to the indices of free variables is maintained explicitly with the term in the environment. Thus, the suspension term shown above would be represented, as it were, as $[\![t_1, 1, nl, (t_2, (nl - nl')) :: nil]\!]$; actually, the old and new embedding levels are needed in this term only for determining the adjustment to the free variables in t_1 with indices greater than the old embedding level, and devices for representing environments encapsulating such an adjustment simplify the actual notation used. The drawback with this approach is that in moving substitutions under abstractions *every* term in the environment is affected. Thus, from a term like $[\![(\lambda\, t_1), 1, nl, (t_2, (nl - nl')) :: nil]\!]$, we must produce one of the form $(\lambda\, [\![t_1, 2, nl + 1, @1 :: (t_2, nl - nl' + 1) :: nil]\!])$. In contrast, using our notation, it is only necessary to add a 'dummy' element to the environment and to make a *local* change to the embedding levels of the overall term.

Both the $\lambda\sigma$-calculus and the suspension notation admit graftable meta variables. The former calculus is known not to preserve strong normalizability [10]. For the suspension notation, this is an open question. We conjecture that it actually does preserve this property.

6 Conclusion

We have exposed the suspension notation in this paper with an eye to its use in metalanguage implementations. Certain questions raised in this discussion need a fuller treatment. In Section 4, we have considered the possibility of utilizing our notation augmented with graftable meta variables in realizing higher-order unification. In reality, this procedure needs to be spelled out in detail and a careful, implementation level comparison with an approach that does not use such variables needs to be done. The benefits of the different treatments of meta variables are likely to depend on the way in which substitutions are generated and, for this reason, the experimentation should also consider special cases of higher-order unification such as that described in [11]. In another direction, it is of interest to manifest the connections between the suspension notation and the $\lambda\sigma$-calculus more completely, possibly via translations between them. Finally, the question of whether or not the suspension notation preserves strong normalizability needs to be settled. We hope to consider some of these aspects in a sequel to this paper.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalization. *Journal of Functional Programming*, 6(5):699–722, 1996.

[3] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[4] R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *To appear in Mathematical Structure in Computer Science*, 2000.

[5] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.

[6] B. Guillaume. The λs_e-calculus does not preserve strong normalisation. *Journal of Functional Programming*, 10(4):321–325, July 2000.

[7] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[8] F. Kamareddine and A. Ríos. Extending the λ-calculus with explicit substitution which preserves strong normalization into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, 1997.

[9] C. Liang and G. Nadathur. Tradeoffs in the intensional representation of lambda terms. In S. Tison, editor, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, July 2002. (To appear).

[10] Paul-André Mellies. Typed λ-calculi with explicit substitutions may not terminate. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, number 902 in Lecture Notes in Computer Science, pages 328–334. Springer, 1995.

[11] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[12] G. Nadathur and D.J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291. Springer-Verlag, July 1999.

[13] G. Nadathur and D.S. Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

[14] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 313–323. ACM Press, September 1998.