Explicit Substitutions in the Reduction of Lambda Terms

Gopalan Nadathur
Computer Science and Engineering
University of Minnesota
4-192 EE/CS Building
200 Union Street S.E.
Minneapolis, MN 55455
gopalan@cs.umn.edu

Xiaochu Qi
Computer Science and Engineering
University of Minnesota
4-192 EE/CS Building
200 Union Street S.E.
Minneapolis, MN 55455
xqi@cs.umn.edu

ABSTRACT

Substitution in the lambda calculus is a complex operation that traditional presentations of beta contraction naively treat as a unitary operation. Actual implementations are more careful. Within them, substitutions are realized incrementally through the use of environments. However, environments are usually not accorded a first-class status within such systems in that they are not reflected into term structure. This approach does not allow the smaller substitution steps to be intermingled with other operations of interest on lambda terms. Various new notations for lambda terms remedy this situation by proposing an explicit treatment of substitutions. Unfortunately, a naive implementation of beta reduction based on such notations has the potential of being costly: each use of the substitution propagation rules causes the creation of a new structure on the heap that is often discarded in the immediately following step. There is, thus, a tradeoff between these two approaches. This paper discusses these tradeoffs and offers an amalgamated approach that utilizes recursion in rewrite rule application but also suspends substitution operations where profitable.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages, constraint and logic languages*; D.3.4 [**Programming Languages**]: Processors—*interpreters*; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*evaluation strategies*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms

Algorithms, Experimentation, Languages, Theory

PPDP'03, August 27–29, 2003, Uppsala, Sweden.

Keywords

Metalanguages, higher-order abstract syntax, lambda calculus, explicit substitution, suspension notation, beta reduction, graph and environment based reduction procedures

1. INTRODUCTION

This paper concerns the treatment of substitution in the implementation of beta reduction over lambda terms. Our interest in considering this operation arises from situations in which lambda terms are used as representational devices as might happen within the context of proof assistants [4, 5, 7, 20], logical frameworks [6, 11] or metalanguages [17, 21]. In these situations, the task of the reduction apparatus is usually to transform a term into a head normal form that then provides the basis for comparison and unification operations. These head normal forms must, however, also pay attention to structure under the scope of abstractions. This is unlike the idea of computation in functional programming languages where the form may be arbitrary under an abstraction, i.e. where weak head normal forms suffice. The form of reduction we are interested in here is, thus, what is referred to as strong reduction in [9].

Traditional presentations of beta contraction take a rather simplistic view of substitution. Thus, this operation is usually presented via a rewrite rule of the form

$$(\lambda x\, t_1)\ t_2 \ \rightarrow\ t_1[x := t_2]$$

where $t_1[x := t_2]$ denotes the expression that is obtained by replacing the free occurrences of x in t_1 by t_2, carrying out the necessary renamings in the process to ensure that binding scopes are properly respected. Unfortunately, this substitution operation is much too complicated to be treated as an atomic one, and actual implementations of the rule have to break it up into a a series of steps that represent a controlled traversal over the structure of t_1. These implementations, in fact, include a treatment of terms with environments or suspended substitutions. Such a treatment has the auxiliary benefit of allowing substitution walks to be combined. Thus, consider the reduction of the term $(\lambda x\, \lambda y\, t_1)\ t_2\ t_3$. It is necessary in this process to substitute t_2 for x and t_3 for y within the structure of t_1. By a process of delaying and combining substitutions, both replacements can be carried out simultaneously, resulting in a savings both in effort and in the amount of new structure created to represent the result of substitution. This kind of an organization of the computation is central to many

lambda calculus interpreters and simplifiers, a primary exemplar being the one described in [2].

In the kinds of situations that are of interest to us, ease in the comparison of terms modulo α-conversion or bound variable renaming is of special significance. A convenient way to factor this matter in is to use the scheme of de Bruijn [3] for eliminating names altogether from terms; this approach obviates α-conversion in equality checking and is also arguably sustainable from the perspective of efficiency in realizing beta reduction [14]. The use of the de Bruijn scheme coupled with the desire to realize strong reduction, however, complicates the treatment of suspended substitutions. In particular, when descending into abstraction contexts, it is necessary to properly modify the correspondence between de Bruijn indices and the terms that are to replace them. Further, indices for free variables within the terms being substituted need also to be modified to account for a new abstraction that is introduced between them and the abstractions eventually binding them. Issues of this kind have been the subject of recent study and, in fact, find treatment in the various explicit substitution notations that have been proposed for the lambda calculus (*e.g.*, see [1, 12, 19]). Such notations can be reflected into the environment based approaches to reduction so as to extend them to treat strong reduction. We indicate how this might be done in this paper.

Environment based reduction procedures, even if guided by an explicit substitution notation, provide only an implicit treatment of substitutions. In particular, the terms eventually produced by such procedures do not contain subparts encoding suspended substitutions. One consequence of this is that opportunities for sharing substitution walks that arise from mixing reduction with other kinds of computation steps are missed. This observation is brought out by an example relevant to formula manipulation. In a higher-order approach to formula representation, the binding character of quantifiers is captured by lambda abstraction. Thus, the formula $\forall x P(x)$ may be encoded by the lambda term $(all\ (\lambda x\, \overline{P(x)}))$, where $\overline{P(x)}$ denotes, recursively, the representation of $P(x)$. Now, relevant computations over formulas may involve the instantiation of universal quantifiers that occur in them. Such instantiations can be realized by recognizing patterns of the form $(all\ P)$ and generating the application $(P\ t)$ where t is the instantiating term; actual substitution in this case is effected via beta contraction. Suppose now that we have at hand a formula encoded by a term of the form $(all\ \ldots (all\ P)\ldots)$, *i.e.*, one in which there are multiple quantifiers scoping over the structure P. It is obviously beneficial to realize all the instantiating substitutions in one walk over P. However, to obtain this effect, it is necessary to delay the actual computation of substitutions not only within a reduction procedure but also over pattern matching steps. Explicit substitution notations provide exactly the means for supporting this kind of processing: they allow substitutions to be represented explicitly in (sub)terms and, hence, to be delayed over other computations. We indicate how this might be done by defining a generalized notion of head normal form and by describing a reduction procedure that exploits this generalization.

The simplest approach to exploiting explicit substitutions is to utilize the rewrite rules that propagate substitutions directly; thus, the reduction procedure becomes one that mainly organizes the application of these rules. A disadvantage of this approach is that it may create many terms that are discarded almost immediately as the next rewrite rule is applied in reduction. The organization of environment based procedures suggests a way to avoid such a potentially profligate use of heap space. Rather than creating new terms with embedded substitutions immediately, these may be represented implicitly through the (environment) parameters and local variables in recursively structured processing. However, when such a process has finally uncovered a generalized head normal form, rather than actually effecting the remaining substitutions, new structures can be created that maintain such substitutions in suspended form. This approach combines the implicit and explicit treatments of substitutions and can accrue the benefits of both. We present a reduction procedure that executes this idea.

One contribution of this paper, then, is to describe three different approaches to the use of explicit substitutions in the implementation of strong reduction.[1] As another contribution, we attempt to quantify the benefits perceived for the different approaches by comparing the heap usage accruing to the creation of new terms under each. This experimental evaluation is carried out by embedding C based realizations of each reduction procedure within the *Teyjus* implementation of λProlog [18] and using the resulting system to run prototypical higher-order logic programs.

The rest of this paper is structured as the follows. In the next section we describe an explicit substitution calculus called the suspension notation [19]; this notation has already been used in two practical systems [18, 22] and is therefore an appealing basis for our study. Sections 3, 4 and 5 then present reduction procedures realizing the three different approaches of interest. These procedures are presented using the SML language both for simplicity of exposition and for concreteness, although the same ideas can be deployed in realizations in any other language as well.[2] Section 6 contains a quantitative comparison of these approaches using, in fact, a C based realization of each. Section 7 concludes the paper.

2. THE SUSPENSION NOTATION

Like other explicit substitution calculi, the suspension notation conceptually encompasses two categories of expressions, one corresponding to terms and the other corresponding to environments that encode substitutions to be performed over terms. In a notation such as the $\lambda\sigma$-calculus [1] that use exactly these two categories, an operation must be performed on an environment expression each time it is percolated inside an abstraction towards modifying the de Bruijn indices in the terms whose substitution it represents. The suspension notation instead allows these adjustments to be carried out in one swoop when a substitution is actually effected rather than in an iterated manner. To support this possibility, this notation includes a third category of expressions called environment terms that encode terms together with the 'abstraction context' they come from.

[1] Some of these approaches may have been used in practical systems in the past; *e.g.*, a variant of one of these is embodied in the *Teyjus* system. Our contribution, however, is in explicitly identifying and contrasting the different possibilities.

[2] We assume in this presentation that our readers are familiar with SML, a tutorial introduction to which may be found, for instance, in [10].

Formally, the syntax of terms, environments and environment terms in our notation is given by the following rules:

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

In these rules, $\langle C\rangle$ represents constants, $\langle V\rangle$ represents instantiatable variables (*i.e.*, variables that can be substituted for by terms), $\langle I\rangle$ is the category of positive numbers and $\langle N\rangle$ is the category of natural numbers. Terms constitute our enrichment of lambda terms. In keeping with the de Bruijn scheme, $\#i$ represents a variable bound by the ith abstraction looking back from the occurrence. An expression of the form $[\![t, ol, nl, e]\!]$, referred to as a *suspension*, is a new kind of term that encodes a term with a 'suspended' substitution: intuitively, such an expression represents the term t with its first ol variables being substituted for in a way determined by e and its remaining bound variables being renumbered to reflect the fact that t used to appear within ol abstractions but now appears within nl of them. Conceptually, the elements of an environment are either substitution terms generated by a contraction or are dummy substitutions corresponding to abstractions that persist in an outer context. However, renumbering of indices may have to be done during substitution, and to encode this the environment elements are annotated by a relevant abstraction level. To be deemed well-formed, suspensions must satisfy certain 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, for each element of the form $@l$ of e it must be the case that $l < j$ and for each element of the form (t', l) of e it must be the case that $l \leq j$.

The expressions in our notation are complemented by a collection of rewrite rules that simulate β-contractions. These rules are presented in Figure 1. We use the notation $e[i]$ in these rules to denote the i^{th} element of the environment. Of the rules presented, the ones labelled (β_s) and (β'_s) generate the substitutions corresponding to the beta contraction rule on de Bruijn terms and the rules (r1)-(r9), referred to as the *reading rules*, serve to actually carry out these substitutions. As an illustration of these roles for the rules, we may consider their use in the reduction of the term

$$((\lambda\,((\lambda\,((\#1\ \#2)\ \#3)))\ t_2))\ t_3),$$

in which t_2 and t_3 denote arbitrary de Bruijn terms. Using the (β_s) rule, this term can be rewritten to

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

We can now use the rules (r6) and (r5) to propagate the substitution, thereby producing

$$\begin{array}{l} ((\lambda\,[\![(\lambda\,((\#1\ \#2)\ \#3)), 2, 1, @0 :: (t_3, 0) :: nil]\!]) \\ \qquad [\![t2, 1, 0, (t_3, 0) :: nil]\!]). \end{array}$$

The (β'_s) rule is applicable to this term and using it yields

$$\begin{array}{l} [\![(\lambda\,((\#1\ \#2)\ \#3)), 2, 0, ([\![t2, 1, 0, (t_3, 0) :: nil]\!], 0) :: \\ \qquad\qquad (t_3, 0) :: nil]\!]. \end{array}$$

Using the rules (r4)-(r7) some number of times now produces

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

Noting the structure of rule (r3), it is easy to see that the term $[\![t_3, 0, 1, nil]\!]$ that appears here represents the result of 'raising' the index of each free variable in t_3 by 1 as is necessitated by its substitution under an abstraction. A similar comment applies to the other embedded suspension. Using the rule (r8), the overall term can be further transformed into

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

The content of this rewriting step is, in effect, to merge a 'renumbering' suspension into the suspension embedded within it. Depending on the particular structures of t_2 and t_3, the reading rules can be applied repeatedly to this term to finally produce a de Bruijn term that results from the original term by contracting the two outermost β-redexes.

The rule (r2) that is included in the collection pertaining to 'reading' instantiatable variables. This rule is actually based on a particular interpretation of such variables: substitutions that are made for them must not contain de Bruijn indices that are captured by external abstractions. This is a common understanding of such variables but not the only one. For example, treating these variables as essentially first-order ones whose instantiation *can* contain free de Bruijn indices provides the basis for lifting higher-order unification to an explicit substitution notation [8].

The (β'_s) rule is redundant to our collection if our sole purpose is to simulate β-contraction. However, as is manifest in the reduction example considered, it is *the* rule in our system for combining substitutions arising from different contractions into one environment and, thereby, for carrying them out in the same walk over the structure of the term being substituted into. This rule will be exploited to this effect in all the reduction procedures we describe. The rule (r8) is also redundant, but, as is again demonstrated in our reduction example, it serves a similar useful purpose in that it allows a reduction to be combined with a renumbering walk after a term has been substituted into a new

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

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

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

(r2) $[\![x, ol, nl, e]\!] \rightarrow x$
provided x is an instantiatable variable

(r3) $[\![\#i, ol, nl, e]\!] \rightarrow \#j$
provided $i > ol$ and $j = i - ol + nl$.

(r4) $[\![\#i, ol, nl, e]\!] \rightarrow \#j$
provided $i \leq ol$ and $e[i] = @l$ and $j = nl - l$.

(r5) $[\![\#i, ol, nl, e]\!] \rightarrow [\![t, 0, j, nil]\!]$
provided $i \leq ol$ and $e[i] = (t, l)$ and $j = nl - l$.

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

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

(r8) $[\![[\![t, ol, nl, e]\!], 0, nl', nil]\!] \rightarrow [\![t, ol, nl + nl', e]\!]$.

(r9) $[\![t, 0, 0, nil]\!] \rightarrow t$

Figure 1: Rewrite rules for the suspension notation

(abstraction) context. In fact, the main uses of rules (r8) and (r9) arise right after a use of rule (r5) and they may therefore be eliminated in favour of the following enhanced versions of (r5):

(r10) $[\![\#i, ol, nl, e]\!] \rightarrow t$,
provided $i \leq ol$, $e[i] = (t, l)$ and $nl = l$.

(r11) $[\![\#i, ol, nl, e]\!] \rightarrow [\![t, ol', nl' + nl - l, e']\!]$,
provided $i \leq ol$, $e[i] = ([\![t, ol', nl', e']\!], l)$, and $nl \neq l$.

This course is followed in our reduction procedures.

The capability of the suspension notation to simulate reduction in the lambda calculus has been shown in [19] in two steps. First, underlying every term in the suspension notation is intended to be a de Bruijn term that is obtained by 'calculating out' the suspended substitutions. The reading rules can be shown to possess properties that support this interpretation: they define a reduction relation that is both strongly normalizing and confluent. It can then be shown that the de Bruijn term t β-reduces to s if and only t can be rewritten to s using the rules in Figure 1.

Our objective in reduction will be to produce head normal forms for terms that can be used, for instance, in subsequent comparison operations. The notion of a head normal form can be lifted as follows to our notation, assuming a willingness to represent suspensions explicitly:

DEFINITION 1. *A suspension term is in head normal form if it has the structure*

$$(\lambda \ \ldots (\lambda \ (\ldots (h \ t_1) \ \ldots \ t_m)) \ldots)$$

where h is a constant, a de Bruijn index or an instantiatable variable. In this case, $t_1, \ldots, t_m$ are called its arguments, h is called its head and n is its binder length.

We shall refer to head normal forms in the de Bruijn notation as head normal forms in the *conventional sense*. Definition 1 extends this notion by allowing the arguments to be suspensions. We shall call the terms described by this definition *generalized* head normal forms. The main utility of these forms arises from the fact that they are also related to the conventional ones at a rewriting level:

THEOREM 2. *Let t be a de Bruijn term. Further suppose that the rules in Figure 1 allow it to be rewritten to a generalized head normal form that has h as its head, n as its binder length and $t_1, \ldots t_m$ as its arguments. Then t has the term*

$$(\lambda \ldots (\lambda \, (\ldots (h \ |t_1|) \ \ldots \ |t_m|)) \ldots)$$

as a head normal form in the conventional sense, where $|t|$ denotes the normal form modulo the reading rules for a suspension term t.

PROOF. See [16] □

3. ENVIRONMENT BASED REDUCTION

We now discuss head normalization procedures that exploit the mechanisms provided by the suspension notations. All the procedures that we consider will be graph-based, *i.e.*, lambda terms will be represented as graphs within them and destructive changes will be used to register, and thus to share, reduction steps.

```
datatype rawterm = const of string
 | bv of int
 | ptr of (rawterm ref)
 | app of (rawterm ref * rawterm ref)
 | lam of (rawterm ref)

type term = (rawterm ref)

datatype eitem = dum of int
 |  bndg of clos * int
and clos = cl of term * int * int * (eitem list)

type env = (eitem list)
```

Figure 2: Type declarations for lambda terms

The first reduction procedure that we present makes use of the new devices in the suspension notation only implicitly. Thus, within this procedure, the idea of suspension will be used to effect the propagation of substitution over terms. However, suspensions will be realized mainly through the structure of recursive calls to the normalization routine; terms that are input to the procedure or that are eventually returned by it will not themselves contain embedded suspensions. In the usual leftmost-outermost reduction control regime that is inherent to head normalization, it is necessary to record *closures*, or terms paired with environments, in environments. The only explicit use of suspensions in our present variety of reduction procedures will occur in the encoding of such closures. Notice that these suspensions appear only at the top-level of terms and also do not persist beyond the reduction process.

Figure 2 provides the datatype declarations in SML that serve to represent terms and closures as needed in the reduction procedure of interest at the moment. We observe that terms are realized as references to appropriate structures in these declarations so as to support a graph based approach to reduction. Complementing this encoding, we use the following two functions to dereference a term and to assign a new value to a given term:

```
fun deref(term as ref(ptr(t))) = deref(t)
  | deref(term) = term

fun assign(t1,ref(ptr(t))) =  assign(t1,t)
  | assign(t1,t2) = t1 := ptr(t2)
```

In the course of reduction, we will often need to look up a value in an environment. The following function is useful for this purpose:

```
fun nth(x::l,1) = x
  | nth(x::l,n) = nth(l,n-1)
```

The head normalization procedure presently of interest has two essential phases. In the first phase, it traces a (generalized) head reduction sequence to produce a head normal form as per Definition 1. Once such a form has been unearthed, a second phase is entered to compute out the effect of suspended substitutions on the arguments. In both these phases, the relevant suspensions are encoded implicitly in the parameters of the procedures. The functions *head_norm1* and *subst* whose definitions appear in Figures 3

```
fun head_norm1(term as ref(const(c)),ol,nl,env,whnf) = (term,0,0,nil)
  | head_norm1(term as ref(bv(i)),ol,nl,env,whnf) =
     if (i > ol)
     then (ref(bv(i-ol+nl)),0,0,nil)
     else
      (fn dum(l) => (ref(bv(nl - l)),0,0,nil)
        | bndg(cl(t,ol',nl',e'),l) =>
            if (l = nl)
            then head_norm1(t,ol',nl',e',whnf)
            else head_norm1(t,ol',nl+nl'-l,e',whnf)
      ) (nth(env, i))
  | head_norm1(term as ref(lam(t)),ol,nl,env,true) = (term,ol,nl,env)
  | head_norm1(term as ref(lam(t)),ol,nl,env,false) =
     let val (t',_,_,_) =
             if (ol = 0) andalso (nl = 0)
             then head_norm1(t,0,0,nil,false)
             else head_norm1(t,ol+1,nl+1,dum(nl)::env,false)
     in  (ref(lam(t')),0,0,nil)
     end
  | head_norm1(term as ref(app(t1,t2)),ol,nl,env,whnf) =
     let val (f,fol,fnl,fe) = head_norm1(t1,ol,nl,env,true)
     in
      (fn ref(lam(t)) =>
           let
             val t2' = cl(t2,ol,nl,env)
             val s' = head_norm1(t,fol+1,fnl,bndg(t2',fnl)::fe,whnf)
             val (t',ol',nl',env') = s'
           in
             (if (ol = 0) andalso (nl = 0) andalso (ol' = 0) andalso (nl' = 0)
              then assign(term,t')
              else ());
             s'
           end
        | f  =>
           if (ol = 0) andalso (nl = 0)
           then (assign(term,ref(app(f,t2))); (term,0,0,nil))
           else
             let val t = app(f,subst(t2,ol,nl,env))
             in (ref(t),0,0,nil)
             end
      ) (deref(f))
     end
  | head_norm1(term as ref(ptr(t)),ol,nl,env,whnf) =
     head_norm1(deref(t),ol,nl,env,whnf)
```

Figure 3: Head normalization with implicit use of suspensions

```
fun subst(term as ref(const(c)),ol,nl,env) =
     term
  | subst(term as ref(app(t1,t2)),ol,nl,env) =
     ref(app(subst(t1,ol,nl,env),
             subst(t2,ol,nl,env)))
  | subst(term as ref(lam(t)),ol,nl,env) =
     ref(lam(subst(t,ol+1,nl+1,dum(nl)::env)))
  | subst(term as ref(bv(i)), ol, nl, env) =
     if i > ol
     then ref(bv(i-ol+nl))
     else
       (fn dum(l) => ref(bv(nl - l))
         | bndg(cl(t,ol',nl',e'),l) =>
             if (l = nl)
             then subst(t,ol',nl',e')
             else subst(t,ol',nl'+nl-l,e')
       ) (nth(env,i))
  | subst(term as ref(ptr(t)),ol,nl,env) =
     subst(deref(t),ol,nl,env)
```

Figure 4: Calculating out suspensions

and 4 serve to implement each of these phases. The invocation of *head_norm1* can occur in one of two modes depending on the value of its last argument that is of boolean type. If this argument is *true*, the intention is to produce a weak head normal form in recognition of the fact that the term to be reduced appears as the function part of an application. This argument being *false*, on the other hand, signals the desire for a (strong) head normal form. The value returned by *head_norm1* will in general be a quadruple that is to be interpreted implicitly as a suspension. In reality, this suspension will be a trivial one in all cases other than when a weak head normal form is computed and the term component of the resulting suspension is an abstraction.

Any given term t may be transformed into head normal form by invoking the procedure *hnorm1* that is defined as follows:

```
fun hnorm1(t) = head_norm1(t,0,0,nil,false)
```

Note that at the end of such a call, t is intended to be a reference to a head normal form of its original value as might be expected in a graph-based reduction scheme. That *hnorm1* correctly realizes this purpose is the content of the following theorem.

THEOREM 3. *Let t be a reference to the representation of a de Bruijn term that has a head normal form. Then hnorm1(t) terminates and, when it does, t is a reference to the representation of a head normal form of the original term.*

PROOF. Only a sketch is provided. The rewriting steps that are carried out by successive invocations of *head_norm1* and *subst* produced by *hnorm1(t)* can be seen to correspond to a generalized head reduction sequence as defined in [16], followed by a sequence of reading rules once a generalized head normal form has been found. The former sequence terminates when t has a head normal form and the reading relation is noetherian. Overall termination is thus assured. The final term must be a head normal form *for* t because of the correctness of the rewrite rules in Figure 1. □

```
datatype rawterm = const of string
  | bv of int
  | ptr of (rawterm ref)
  | lam of (rawterm ref)
  | app of (rawterm ref) * (rawterm ref)
  | susp of (rawterm ref)*int*int*(eitem list)
and eitem = dum of int
  | bndg of (rawterm ref) * int

type env = (eitem list)

type term = (rawterm ref)
```

Figure 5: Type declarations for suspension terms

The procedures that we have presented here create more new terms than are strictly necessary. As an example, given an application, there is no need to create a new version of it if its subcomponents are unaffected by the reduction process. This kind of improvement can be obtained by including a flag in the returned value that indicates whether or not the incoming term has been changed. We have omitted this optimization here for the sake of simplicity of discussion. A similar effect can also be obtained with the reduction procedures that use suspensions explicitly by associating a closedness annotations with terms [16] and this simplification therefore does not affect the comparisons we make between different approaches in a later section.

4. EXPLICIT USE OF SUSPENSIONS

We now consider the possibility of using suspensions directly in term reduction. To obtain this effect we need to first enhance the structure of our term representation in SML. Figure 5 provides the new datatype declarations for this purpose. The main difference between these declarations and the ones in Figure 2 is that suspensions are now formally accepted as one possibility for terms. The suspension notation also allows terms of any arbitrary form, as opposed to only closures, to appear in environments and the structure of environment items is also modified to reflect this possibility. Terms are, once again, realized as references to support graph based reduction and we assume that the functions *assign* and *deref* are available as before to support destructive updates and the concomitant dereferencing of pointers.

In the present version of reduction, our intention is to use the rewrite rules more or less directly in trying to produce a head normal form. This approach involves the creation on the heap of representations for all the new structures that appear on the right hand side of a rule immediately on the application of the rule. Thus, suppose that at a certain point in computation, we use the rule

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

for propagating substitutions over applications. In the mode that we are presently considering, we will create the new structures $[\![t_1, ol, nl, e]\!]$ and $[\![t_2, ol, nl, e]\!]$ and destructively update the term on the lefthand side with an application formed out of these two pieces before proceeding to the next step in reduction. The other facet of the present approach is, of course, that we will not shy away from leaving suspensions in terms when it is unnecessary to evaluate them. In

```
fun beta_contract(term,t1 as ref(susp(t3,ol,nl,dum(nl1) :: e)),t2) =
      if nl = nl1 + 1
      then term := susp(t3,ol,nl1, bndg(t2,nl1) :: e)
      else term := susp(t1,1,0,[bndg(t2,0)])
  | beta_contract(term,t1,t2) = term := susp(t1,1,0,[bndg(t2,0)])

fun lazy_read(term as ref(susp(t,ol,nl,env))) = lazy_read_aux(term,deref(t),ol,nl,env)
  | lazy_read(_) = ()
and
    lazy_read_aux(t1,t2 as ref(const(_)),_,_,_) = t1 := !t2
  | lazy_read_aux(t1,ref(bv(i)),ol,nl,env) =
      if i > ol
      then t1 := bv(i - ol + nl)
      else
       ((fn dum(l) => t1 := bv(nl - l)
          | bndg(t2,l) =>
              ( if (nl = l)
                then assign(t1,t2)
                else
                  ((fn ref(susp(t3,ol',nl',e')) => t1 := susp(t3,ol',nl'+ nl - l,e')
                     | t => t1 := susp(t,0,nl - l,nil)
                  ) (deref t2));
                (lazy_read t1)
              )
        ) (nth (env,i)))
  | lazy_read_aux(t1,ref(app(t2,t3)),ol,nl,env) =
      t1 := app(ref(susp(t2,ol,nl,env)),ref(susp(t3,ol,nl,env)))
  | lazy_read_aux(t1,ref(lam(t2)),ol,nl,env) =
      t1 := lam(ref(susp(t2,ol+1,nl+1,dum(nl)::env)))
  | lazy_read_aux(t1,t,ol,nl,env) =
      (lazy_read(t) ; lazy_read_aux(t1,deref(t),ol,nl,env))

fun head_norm2(term as ref(app(t1,t2)),whnf) =
      (head_norm2(t1,true) ;
       (fn ref(lam(t)) => (beta_contract(term,t,t2); head_norm2(term,whnf))
         | _  => ()) (deref t1))
  | head_norm2(ref(lam(t)),false) = head_norm2(t,false)
  | head_norm2(term as ref(susp(_,_,_,_)),whnf) = (lazy_read(term) ; head_norm2(term,whnf))
  | head_norm2(term as ref(ptr(t)),whnf) = (head_norm2(t,whnf) ; assign(term,t))
  | head_norm2(_,_) = ()

fun hnorm2(t) = head_norm2(t,false)
```

Figure 6: Head normalization using suspensions and immediate rewriting

particular, our reduction procedure will return generalized head normal forms for terms and will not need to calculate out the effects of substitutions at the end as was the case for the earlier reduction procedure. A consequence of this approach is, of course, that our procedure may encounter suspensions during processing and it should therefore include mechanisms for incrementally 'unravelling' these kinds of terms.

Figure 6 presents the definition of a function *head_norm2* that realizes this overall approach to reduction. This procedure can, once again, be invoked in one of two modes depending the value of its second argument that is of boolean type. If this value is *true*, this signals the desire for a weak head normal form of the term whose representation is referenced by the first argument. A strong head normal form is to be produced otherwise. An auxiliary procedure called *lazy_read* is used in this code to realize the incremental exposure of a non-suspension structure from a suspension whenever a term of this category is encountered during normalization. Finally, the function *beta_contract* that appears in this code has the purpose of determining which of the (β_s) and (β'_s) rules is appropriate to use when a beta redex has been discovered and of effecting the corresponding rewriting step.

The intended use of the function *hnorm_norm2* is as follows. Given a reference t to the representation of a lambda term that we wish to transform to head normal form, we evaluate the expression *hnorm_norm2(t,false)*; this is, in fact, the interface that is provided by the function *hnorm2*. Upon return from this invocation, the expectation is that t is a reference to a head normal form of the input term. That *head_norm2* correctly realizes this purpose is the content of the following theorem.

THEOREM 4. *Let t be a reference to the representation of a suspension term that translates via the reading rules to a de Bruijn term with a head normal form. Then hnorm2(t) terminates and, when it does, t is a reference to the representation of a generalized head normal form of the original term.*

PROOF. Once again we provide only a sketch. The successive invocations of *head_norm2* and *lazy_read* that are engendered by the evaluation of *hnorm2(t)* produce a sequence of rewrites that correspond to what has been identified in [16] as a generalized head reduction sequence relative to the term represented by t. It has also been shown in [16] that every such sequence must terminate if the underlying de Bruijn term has a head normal form. This argument guarantees termination of *hnorm2(t)*. The fact that it is the rewrite rules in Figure 1 that are implemented now assures us that it is a normal form of the term corresponding to t that is found. □

5. COMBINING IMPLICIT AND EXPLICIT USES OF SUSPENSIONS

The two previous reduction procedures have complementary benefits and drawbacks. The procedure that uses suspensions explicitly adopts a somewhat naive approach to rewriting and ignores the natural flow of control present in reduction. Thus, consider again the rule for propagating substitutions over applications:

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

An eager creation of the structures $[\![t_1, ol, nl, e]\!]$, $[\![t_2, ol, nl, e]\!]$ and the application on the righthand side has the potential for using heap space unnecessarily: the very next steps may require the first of these suspensions to be rewritten and, a few steps later, it is possible that the outer application itself may be recognized as a beta redex. The procedure that uses suspensions only implicitly, avoids this problem by maintaining information needed for reduction in the recursion stack and committing structures to heap only when these are known to be necessary. However, this procedure does not allow any suspensions into terms at all. There are problems with this structure as well. In particular, it becomes impossible under this approach to delay substitutions over other computations and, hence, to combine substitution walks that arise from beta redexes that are created in the course of other processing.

Fortunately, an amalgamation of the two approaches can be attempted. The essential idea is to follow the basic processing regime of the environment based reduction procedure but, in the end, when a generalized head normal form has been exposed, to leave the uncomputed substitutions in the form of suspensions.

In order to implement this approach it is, of course, necessary to use the richer representation of terms that includes an encoding of suspensions. Assuming the datatype declarations in Figure 5, a collection of ML functions that realize the proposed idea are shown in Figure 7. Let t be a reference to the representation of a term the we wish to head normalize. The function invocation *hnorm3(t)* is intended to achieve this effect: upon return from this call, t should be a reference to a generalized head normal form of the original term.

The main work in reduction is actually performed by the function *head_norm3*. This function has a structure that is in most respects identical to the environment based procedure *head_norm1* that we have seen earlier. There are, in fact, only two significant differences. The first of these relates to the processing of an application when this has been recognized to be a (left) part of a head normal form. In such a situation, rather than computing out the effects of a nontrivial substitution, the argument part of the application is encapsulated as a suspension. The second difference arises from the fact that the new procedure must be prepared to also process suspensions. It is interesting to note that, in order to preserve the ability to commit structures to heap only when necessary, the embedded suspension needs to be processed first in this case. This order is different from the one used by the reduction procedure of the previous section that commits structures to heap eagerly. However, the progression of reduction steps is still encompassed by the generalized notion of head reduction sequence defined in [16].

The correctness of *hnorm3* is the content of the following theorem whose proof is similar to those of Theorems 3 and 4.

THEOREM 5. *Let t be a reference to the representation of a suspension term that translates via the reading rules to a de Bruijn term with a head normal form. Then hnorm3(t) terminates and, when it does, t is a reference to the representation of a generalized head normal form of the original term.*

```
fun make_explicit(t,0,0,nil) = t
  | make_explicit(ref(lam(t)),ol,nl,env) = ref(lam(ref(susp(t,ol+1,nl+1,dum(nl)::env))))

fun head_norm3(term as ref(const(c)),ol,nl,env,whnf) = (term,0,0,nil)
  | head_norm3(term as ref(bv(i)),0,0,nil,whnf) = (term,0,0,nil)
  | head_norm3(term as ref(bv(i)),ol,nl,env,whnf) =
      if (i > ol) then (ref(bv(i-ol+nl)),0,0,nil)
      else
       (fn dum(l) => (ref(bv(nl-l)),0,0,nil)
         | bndg(t,l) =>
             if (nl = l) then head_norm3(t,0,0,nil,whnf)
             else
               ((fn ref(susp(t2,ol',nl',e')) => head_norm3(t2,ol',nl'+ nl - l,e',whnf)
                  | t => head_norm3(t,0,nl-l,nil,whnf)
                ) (deref t))
       ) (nth(env, i))
  | head_norm3(term as ref(lam(t)),ol,nl,env,true) = (term,ol,nl,env)
  | head_norm3(term as ref(lam(t)),ol,nl,env,false) =
      let val (t',ol',nl',env') =
            if (ol = 0) andalso (nl = 0)
            then head_norm3(t,0,0,nil,false)
            else head_norm3(t,ol+1,nl+1,dum(nl)::env,false)
      in (ref(lam(t')),ol',nl',env')
      end
  | head_norm3(term as ref(app(t1,t2)),ol,nl,env,whnf) =
      let val (f,fol,fnl,fenv) = head_norm3(t1,ol,nl,env,true)
      in
        (fn ref(lam(t)) =>
              let val t2' = if (ol = 0) andalso (nl = 0)
                            then t2
                            else ref(susp(t2,ol,nl,env))
                  val s = head_norm3(t,fol+1,fnl,bndg(t2',fnl)::fenv,whnf)
                  val (t',ol',nl',env') = s
              in
                (if (ol = 0) andalso (nl = 0) andalso (ol' = 0) andalso (nl' = 0)
                 then assign(term,t')
                 else ());
                s
              end
          | f => if (ol = 0) andalso (nl = 0)
                 then (assign(term,ref(app(f,t2))); (term,0,0,nil))
                 else (ref(app(f,ref(susp(t2,ol,nl,env)))),0,0,nil)
        ) (deref(f))
      end
  | head_norm3(term as ref(susp(t,ol,nl,env)),ol',nl',env',whnf) =
      let val s =  head_norm3(t,ol,nl,env,whnf)
          val t' = (make_explicit s)
      in
        ( assign(term,t');
          if (ol' = 0) andalso (nl' = 0)
          then s
          else head_norm3(term,ol',nl',env',whnf)
        )
      end
  | head_norm3(term as ref(ptr(t)),ol,nl,env,whnf) =
      head_norm3(deref(t),ol,nl,env,whnf)

 fun hnorm3(t) = head_norm3(t,0,0,nil)
```

Figure 7: Head normalization using suspensions implicitly and explicitly

	implicit suspensions	explicit suspensions	combination approach
[typeinf]	20,834,989	11,044,078	4,508,664
	30,478,132	26,982,390	9,447,584
[compiler]	4,565,938	777,803	331,973
	6,117,710	1,866,979	693,387
[church]	227,271	214,334	148,970
	411,368	500,448	236,158
[hilbert]	220,358	27,263	11,932
	356,882	69,086	21,535

Figure 8: Heap usage for different reduction approaches

6. COMPARISONS

We now consider a quantification of the relevance in practice of the intuitions underlying the various reduction procedures discussed in the earlier sections. The higher-order logic programming language λProlog provides an excellent framework for conducting such a study. This language employs lambda terms as a means for realizing higher-order approaches to the processing of syntactic structure. Thus, within it, lambda terms are available for use in representing objects whose understanding embody binding notions, and operations such as higher-order unification and reduction can be utilized for manipulating such representations in logically meaningful ways. By running a variety of actual λProlog programs and collecting suitable data over these, we can therefore obtain an understanding of the impact of the different approaches to reduction.

We have carried out the described idea by taking advantage of a compiler and abstract machine based implementation of λProlog called *Teyjus*. This system, that is implemented in the C language, supports a low-level encoding of lambda terms based on the suspension notation. Moreover, reduction computations within it are isolated to a head normalization procedure that is invoked at relevant points to produce terms in a form that is appropriate for subsequent comparison operations. As a basis for our study, we have implemented three different versions of this normalization procedure following the lines of discussion in this paper, and we have metered these to collect information about the number of heap cells created over the entire duration of any given user program.[3]

The data that we provide here has been obtained by running the following representative user programs:

- *[typeinf]* A program that infers type schemes for ML-like programs based on the Damas-Milner system.
- *[compiler]* A compiler for a small imperative language [13].
- *[church]* A program that involves arithmetic computations with Church numerals and associated combinators.
- *[hilbert]* A λProlog encoding of Hilbert's Tenth Problem.

The first two programs exemplify what might be called the L_λ style of programming [15]. Computations in this class proceed by first dispensing with all abstractions in lambda term encodings of objects, then carrying out a first-order style analysis over the remaining structure and eventually abstracting out the new constants. As an programming idiom, this is a popular one amongst λProlog, Elf and Isabelle users and, in fact, arguably the most important case to consider in performance assessments. The third program represents a situation in which mainly normalization computations are involved over lambda terms and the last program includes also cases of genuine higher-order unification calculations.

Figure 8 tabulates information that we have gathered using the different implementations of head normalization over this collection of examples. The two rows corresponding to each λProlog program indicate, respectively, the number of internal term nodes and the number of heap cells created in the course of executing the program; these figures are distinct because the number of heap cells needed for a given term node varies in the *Teyjus* implementation depending on the type of the node. The columns are to be understood as follows: *implicit suspensions* corresponds to the reduction scheme where suspensions are recorded only in the structure of recursive procedure calls, *explicit suspensions* corresponds to the approach that explicitly realizes each rewrite rule in Figure 1 and the *combination approach* represents the amalgamation of the two other ones.

The data that we have presented indicate a significant superiority, especially in the case of the L_λ-style of programs, in terms of reduced heap usage for an approach that utilizes suspension explicitly in term representation. These benefits are also further enhanced by an exploitation of the processing structure to delay the commitment of the effects of rewriting to the heap.

7. CONCLUSION

We have examined different approaches to using explicit substitutions in reduction computations in this paper. Most simplifiers for the lambda calculus actually take advantage of this notion in spirit, as is manifest in their use of environments and closures. However, it is often believed that an intrinsic use of explicit substitutions—in particular, a reflection of substitution encodings into term structure and the creation of such terms during reduction—can be costly in terms of space usage. There is some justification to this belief when the entire structure of the term to be normalized is known at the outset and when the reduction process is invoked only once over this term. However, there are many metalanguages, logical frameworks and proof assistants where computations over lambda terms have a significantly different character. In these systems, many substitutions may have to be performed into the same subcomponent of a given object in the course of a larger calculation. These substitutions are effected by creating and contracting at different points of time β-redexes that span over the relevant structure. An explicit rendition of substitutions appears to be the only way to combine the effecting of such substitutions and, hence, to avoid redundant structure creation in these cases. Experiments that we have conducted show that

[3]Another important factor to assess is the impact on processing time of the different strategies. However, a systematic study of this aspect must optimize the reduction implementation to the particular strategy used and must also include garbage collection costs. A consideration of these issues is beyond the scope of the present study.

this can be a significant factor in practical situations. The delaying of substitutions even with a naive realization of rewriting can, as we have seen, substantially reduce the demand for heap space. Moreover, we have described a more sophisticated approach to rewriting that exploits the processing structure to avoid the creation of redundant intermediate terms towards further reducing heap usage.

Our focus in this work has been mainly on a comparison of space usage and the elimination of redundant structure creation. Another important factor to consider is the time efficiency of each of the reduction approaches. The procedure based on the naive view of rewriting is the simplest to realize and the *Teyjus* system, in fact, embodies an iterative rendition of this procedure using a term stack. We are currently examining optimized implementations of the other reduction strategies and also a way in which to take garbage collection costs into account. Once a treatment of these aspects has been determined, it will be meaningful to obtain, and contrast, time measurements. A different aspect that is relevant to study concerns the compiled realization of reduction. Recent work relative to the *Coq* system has shown how to use compilation assuming eager reduction and substitution strategies to obtain substantial speedups in comparison with the existing interpretive approach [9]. The examples considered in this study seem to be ones where the terms to be normalized are available in complete form at the beginning of the computation. As we have argued, this situation is different from what is encountered in metalanguages such as λProlog and Elf. It is, therefore, of interest to see if explicit substitutions can be built into a compilation model towards harnessing the benefits of laziness in substitution over and above those of compilation in these contexts.

8. ACKNOWLEDGEMENTS

This work was supported in part by Grant CCR-0096322 from the NSF and by a Grant in Aid of Research from the University of Minnesota. We are grateful to the reviewers for their comments that were helpful in improving content and presentation.

9. 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] L. Aiello and G. Prini. An efficient interpreter for the lambda-calculus. *The Journal of Computer and System Sciences*, 23:383–425, 1981.

[3] N. 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] N. Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[5] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[6] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[7] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

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

[9] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 235–246, Pittsburgh, October 2002.

[10] R. Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, University of Edinburgh, November 1986. Revised by Nick Rothwell, January 1989, with exercises by Kevin Mitchell.

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[12] 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.

[13] C. Liang. Compiler construction in higher order logic programming. In *4th International Symposium on Practical Aspects of Declarative Languages*, pages 47–63. Springer Verlag LNCS No. 2257, 2002.

[14] C. Liang and G. Nadathur. Tradeoffs in the intensional representation of lambda terms. In S. Tison, editor, *Conference on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 192–206. Springer-Verlag, July 2002.

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

[16] G. Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), 1999.

[17] G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.

[18] 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.

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

[20] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[21] F. Pfenning. Elf: A meta-language for deductive

systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[22] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323. ACM Press, September 1998.