Choices in Representation and Reduction Strategies for Lambda Terms in Intensional Contexts

Chuck Liang,[1] Gopalan Nadathur[2] and Xiaochu Qi[3]

Abstract

Higher-order representations of objects such as programs, proofs, formulas and types have become important to many symbolic computation tasks. Systems that support such representations usually depend on the implementation of an intensional view of the terms of some variant of the typed lambda calculus. New notations have been proposed for the lambda calculus that provide an excellent basis for realizing such implementations. There are, however, several choices in the actual deployment of these notations the practical consequences of which are not currently well understood. We attempt to develop such an understanding here by examining the impact on performance of different combinations of the features afforded by such notations. Amongst the facets examined are the treatment of bound variables, eagerness and laziness in substitution and reduction, the ability to merge different structure traversals into one and the virtues of annotations on terms that indicate their dependence on variables bound by external abstractions. We complement qualitative assessments with experiments conducted by executing programs in a language that supports an intensional view of lambda terms while varying relevant aspects of the implementation of the language. Our study provides insights into the preferred approaches to representing and reducing lambda terms and also exposes characteristics of computations that have a somewhat unanticipated effect on performance.

1 Introduction

We are concerned in this paper with the representation and manipulation of lambda terms in situations where they are employed as devices for encoding complex syntactic objects. Our interest in this issue arises from the implementation of programming systems such as proof assistants [Bru80, CAB$^+$86, Tea02, NPW02], logical frameworks [CH88, HHP93] and metalanguages [NM88, PS99]. Within these systems, the terms of a chosen lambda calculus are used as data structures, with abstraction in these terms serving to encode the binding notions present in objects such as formulas, programs and proofs, and the attendant β-reduction operation capturing substitution computations. Furthermore, logical manipulations over the encoded objects often involve analyses of their representations using a form of unification that incorporates equality under the lambda conversion rules. Finally,

[1] Department of Computer Science, Hofstra University, Hempstead, NY 11550, cscccl@hofstra.edu.

[2] Department of Computer Science and Engineering and Digital Technology Center, University of Minnesota, 4-192 EE/CS Building, 200 Union Street S.E., Minneapolis, MN 55455, gopalan@cs.umn.edu.

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

the actual execution of such manipulations usually involves some form of search, realized, for instance, through a depth-first search regimen complemented with backtracking.

Lambda terms also underlie functional programming languages and their treatment in this context is usually based on their compilation into forms whose only required relationship to the original terms is that they both reduce to the same values. Such a translation is, of course, not acceptable in the situation in which we are presently interested. Instead, a representation is required that provides access at runtime to the *form* of a term and that facilitates comparisons between terms based on this structure. More specifically, these comparison operations generally ignore bound variable names and equality modulo α-conversion must therefore be easy to recognize. Further, comparisons of terms must factor in the β-conversion rule and, to support this, an efficient implementation of β-contraction must be provided. An essential component of β-contraction is a substitution operation over terms. Building in a fine-grained control over this operation has been thought to be useful. While such control can be realized in principle by encoding substitutions in environments, care is needed in how exactly this is done because the comparison of terms can involve the propagation of substitutions and the contraction of redexes inside the contexts of abstractions.

The representational issues outlined above have been examined in the past and approaches to dealing with them have also been described. A well-known solution to the problem of identifying two lambda terms that differ only in the names chosen for bound variables is, for instance, to transform them into a nameless form using a scheme due to de Bruijn [Bru72]. Similarly, several new notations for the lambda calculus have been described in recent years that have the purpose of making substitutions explicit (*e.g.*, see [ACCL91, BBLRD96, KR97, NW98]). However, the actual manner in which all these devices should be deployed in a practical context is far from clear. In particular, there are tradeoffs involved with different choices and determining the precise way in which to make them requires experimentation with an actual system: the operations on lambda terms that impact performance are ones that arise dynamically and they are notoriously difficult to predict from the usual static descriptions of computations.

We seek to illuminate this empirical question in this paper. We base our investigation on computations carried out within the language λProlog [NM88] that employs lambda terms as data structures and that embodies many of the kinds of operations over these terms that we have described above. The specific vehicle for our study is a new implementation of this language [NM99] that isolates several choices in term representation, permitting them to be varied and their impact to be quantified. As we argue in detail in later sections, both the language and the implementation context that we have chosen provide us with a framework for investigation that actually encompasses a wide variety of relevant systems. Now, in this concrete setting, we employ a mixture of actual experiments and analyses motivated by these experiments to gain an understanding on the following issues:

1. The benefits and drawbacks of adopting the de Bruijn notation for lambda terms.

2. The relative merits of a spectrum of reduction strategies, ranging from ones that fully normalize terms each time their structure is examined to ones that expose structure

in an incremental, demand-driven manner.

3. The importance of an explicit treatment of substitutions that allows for the combination of ones arising from the contraction of multiple β-redexes into a single environment.

4. The usefulness of a scheme for annotating (sub)terms that indicates their dependence on variables bound by external abstractions.

From this examination we determine that both the de Bruijn scheme and the ability to combine substitutions are important in a practical realization. Our conclusions concerning the other issues are more mixed: different reduction strategies when combined with other features can lead to comparable performance and the impact of annotations depends on the reduction strategy in use. However, we gain significant insight from trying to understand behaviour even in these cases, one that is, in fact, valuable to choosing the right combination of features in practice.

The rest of this paper is structured as follows. In the next section, we describe the explicit substitution notation that underlies the *Teyjus* system and indicate why this is a good choice for our study. In Section 3, we outline the structure of λProlog computations, categorize these into conceptually distinct classes and describe specific programs in each class that we use to make measurements. The following four sections discuss, in turn, the impacts of choices in reduction strategies, the treatment of bound variables, the ability to combine substitutions and annotations. Section 8 summarizes our findings and indicates possible extensions to our study. The paper also has two appendices that substantiate the reduction approaches that we examine in Section 4.

2 The Explicit Treatment of Substitution

Theoretical presentations of the lambda calculus often treat the substitution necessitated by reduction as an atomic operation. Actual implementations have to take a much more realistic view. The effecting of substitution involves a structure traversal similar to that needed for finding and contracting β-redexes. This must be taken into account in properly assessing computational costs. Functional programming language implementations in fact combine all these walks over structure by accumulating substitutions that need to be performed into an environment. These implementations assume that there is no need to look inside abstractions in terms, thereby obviating variable renaming and making it possible to use a simple form of environments. When an abstraction is actually encountered, the meta-level device of a closure is used to suspend the actual performance of the substitution.

The assumption that it is unnecessary to look under abstractions is, unfortunately, not really applicable in the situation where lambda terms are used for representation. For example, consider the task of determining whether the two terms

$$((\lambda\,(\lambda\,(\lambda\,((\#3\ \#2)\ s))))\ (\lambda\,\#1)) \quad \text{and} \quad ((\lambda\,(\lambda\,(\lambda\,((\#3\ \#1)\ t))))\ (\lambda\,\#1))$$

are equal. We use the de Bruijn notation for lambda terms here, writing $\#i$ to represent a variable occurrence bound by the ith abstraction looking outwards.[4] Now, in ascertaining that these terms are not equal, it is necessary to propagate substitutions generated by β-contractions *under* abstractions and also to contract redexes embedded *inside* abstractions. The idea of an environment needs to be carefully adapted to yield a delaying mechanism relative to these requirements. For instance, if a term of the form $((\lambda\, t)\ s)$ is embedded within abstractions, it is to be expected that $(\lambda\, t)$ contains free variables. Hence, if the result of contracting this term is to be encoded by the term t and an 'environment', then the environment must record not just the substitution of s for the first free variable but also the 'decrementing' of the indices corresponding to all the other free variables. Similar observations can be made about propagating substitutions under abstractions.

Explicit substitution calculi extend the notion of environments to yield a treatment of these issues. Moreover, these calculi reflect suspended substitutions directly into term structure, thereby offering a flexibility in ordering computations. We study the benefits of this flexibility in this paper based on a particular calculus known as the *suspension notation* [NW98]. We outline this notation in this section to facilitate this discussion. Although our empirical study must utilize a particular system, the suspension notation is general enough for our observations to eventually be calculus independent. We make this point below by contrasting this system with the other explicit substitution calculi in existence.

2.1 The Suspension Notation

The suspension notation conceptually encompasses two categories of expressions, one corresponding to terms and the other corresponding to environments that encode substitutions. In a notation such as the $\lambda\sigma$-calculus [ACCL91] that uses 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 uses a global mechanism for recording these adjustments so that they can be effected at one time, when a substitution is actually made, 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. The notation additionally incorporates a system for annotating terms to indicate whether or not they contain externally bound variables.

Formally, the syntax of terms, environments and environment terms of the suspension notation are given by the following rules:

$$\begin{array}{l}
\langle T\rangle ::= \langle C\rangle \mid \langle V\rangle \mid \#\langle I\rangle \mid (\langle T\rangle\ \langle T\rangle)_{\langle A\rangle} \mid (\lambda_{\langle A\rangle}\ \langle T\rangle) \mid [\![\langle T\rangle, \langle N\rangle, \langle N\rangle, \langle E\rangle]\!]_{\langle A\rangle} \\
\langle E\rangle ::= nil \mid \langle ET\rangle :: \langle E\rangle \qquad \langle ET\rangle ::= @\langle N\rangle \mid (\langle T\rangle, \langle N\rangle) \qquad \langle A\rangle ::= o \mid c
\end{array}$$

In these rules, $\langle C\rangle$ represents constants, $\langle V\rangle$ represent 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

[4] We note that our initial choice of the de Bruijn notation is orthogonal to other ones that we discuss concerning representation and reduction.

the category of natural numbers. Terms constitute our enrichment of lambda terms. As already noted, $\#i$ corresponds to the de Bruijn rendition of bound variable occurrences. An expression of the form $[\![t, ol, nl, e]\!]_o$ or $[\![t, ol, nl, e]\!]_c$, 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 the environment 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.[5] 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 $[\![t, i, j, e]\!]_o$ or $[\![t, i, j, e]\!]_c$, 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 $l \leq j$. A final point to note about the syntax of our expressions is that all non-atomic terms are annotated with either c or o. The former annotation indicates that the term in question does not contain any variables bound by external abstractions and the latter is used when either this is not true or when enough information is not available to determine that it is. Our well-formedness criteria for terms additionally require that the annotations they carry be consistent with these interpretations.

The expressions in our notation are complemented by a collection of rewrite rules that simulate β-contractions. These rules are presented in Figure 1. The symbols v and u that are used for annotations in these rules are schema variables that can be substituted for by either c or o. We also use the notation $e[i]$ 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 β-contraction rule on de Bruijn terms and the rules (r1)-(r12), 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

$$((\lambda_c\, ((\lambda_o\, (\lambda_o\, ((\#1\ \#2)_o\ \#3)_o))\ t_2)_o)\ t_3)_c,$$

in which t_2 and t_3 denote arbitrary (annotated) de Bruijn terms; note that the consistency of annotations dictates that t_3 be a constant, an instantiatable variable or a complex term carrying the annotation c. Now, using the (β_s) rule, this term can be rewritten to

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

An interesting observation here is that the rewriting step preserves the content of the initial annotations. We can now use the rules (r6) and (r7) to propagate the substitution, thereby producing

$$((\lambda_c\, [\![(\lambda_o\, ((\#1\ \#2)_o\ \#3)_o), 2, 1, @0 :: (t_3, 0) :: nil]\!]_o)\ [\![t_2, 1, 0, (t_3, 0) :: nil]\!]_c)_c.$$

[5] Thus, one may read ol as 'old embedding level' and nl as 'new embedding level.'

(β_s) $((\lambda_u\, t_1)\ t_2)_v \rightarrow \llbracket t_1, 1, 0, (t_2, 0) :: nil \rrbracket_v$

(β'_s) $((\lambda_u\, \llbracket t_1, ol+1, nl+1, @nl :: e \rrbracket_o)\ t_2)_v \rightarrow \llbracket t_1, ol+1, nl, (t_2, nl) :: e \rrbracket_v$

(r1) $\llbracket c, ol, nl, e \rrbracket_u \rightarrow c$
provided c is a constant

(r2) $\llbracket x, ol, nl, e \rrbracket_u \rightarrow x$
provided x is an instantiatable variable

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

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

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

(r6) $\llbracket (t_1\ t_2)_u, ol, nl, e \rrbracket_v \rightarrow (\llbracket t_1, ol, nl, e \rrbracket_v\ \llbracket t_2, ol, nl, e \rrbracket_v)_v$.

(r7) $\llbracket (\lambda_u\, t), ol, nl, e \rrbracket_v \rightarrow (\lambda_v\, \llbracket t, ol+1, nl+1, @nl :: e \rrbracket_o)$.

(r8) $\llbracket (t_1\ t_2)_c, ol, nl, e \rrbracket_u \rightarrow (t_1\ t_2)_c$.

(r9) $\llbracket (\lambda_c\, t), ol, nl, e \rrbracket_u \rightarrow (\lambda_c\, t)$.

(r10) $\llbracket \llbracket t, ol, nl, e \rrbracket_c, ol', nl', e' \rrbracket_u \rightarrow \llbracket t, ol, nl, e \rrbracket_c$.

(r11) $\llbracket \llbracket t, ol, nl, e \rrbracket_o, 0, nl', nil \rrbracket_o \rightarrow \llbracket t, ol, nl + nl', e \rrbracket_o$.

(r12) $\llbracket t, 0, 0, nil \rrbracket_u \rightarrow t$

Figure 1: Rule schemata for rewriting annotated terms

The annotation on the embedded suspension $\llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket_c$ indicates that it must not contain variables bound by external abstractions, something that is necessary for the annotations on the original term to be correct. The (β'_s) rule is applicable to the term at this stage and using it yields

$$\llbracket (\lambda_o\, ((\#1\ \#2)_o\ \#3)_o), 2, 0, (\llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket_c, 0) :: (t_3, 0) :: nil \rrbracket_c.$$

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

$$(\lambda_c\, ((\#1\ \llbracket \llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket_c, 0, 1, nil \rrbracket_o)_o\ \llbracket t_3, 0, 1, nil \rrbracket_o)_o).$$

Noting the structure of rule (r3), it is easy to see that the term $\llbracket t_3, 0, 1, nil \rrbracket_o$ that appears here represents the result of ‘raising’ the index of each externally bound variable in t_3 by 1 as is necessitated by its substitution under an abstraction. A similar comment applies to the other embedded suspension. Actually, neither of the internal terms can contain such a variable and so this renumbering is vacuous. A realization of this fact is encoded in the rules (r8)-(r10) that allow the overall term to be simplified to

$$(\lambda_c\, ((\#1\ \llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket_c)_o\ t_3)_o).$$

The reading rules can now be applied repeatedly to the remaining suspension, producing eventually an (annotated) de Bruijn term that results from the original term by contracting the two outermost β-redexes.

2.2 Some Formal Properties

Interest in the suspension notation derives ultimately from its ability to simulate reduction in the conventional lambda calculus. This capability has been shown in [Nad99] 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 and erasing the annotations. 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. Given a term t, we shall use the notation $|t|$ below to denote the de Bruijn term obtained by annotation erasure from the the normal form of t modulo the reading rules. Now, in the second step, it can then be shown that the suspension term t reduces to s using the rules in Figure 1 if and only if $|t|$ β-reduces to $|s|$. As a particular case of this, each β-contraction in the conventional setting can be realized by a use of the (β_s) rule followed by a sequence of reading steps.

Head normal forms play an important role in the comparison of lambda terms. The following definition recalls the conventional notion and also lifts it to the suspension notation:

Definition 1 *A de Bruijn 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 and there are n abstractions at the front of the term; by a harmless abuse of notation, we permit n and m to be 0 in this presentation. Given such a form, $t_1, \ldots, t_m$ are called its arguments, h is called its head and n is its binder length. Head normal forms are extended to the suspension notation by allowing for annotations and permitting the arguments to be arbitrary suspension terms.

Actual comparison of two terms usually proceeds by reducing them to head normal form, matching their binders and heads and then comparing their arguments. The extension of head normal forms to the suspension notation permits substitutions over the arguments to be delayed until we actually need to compare them. The legitimacy of this approach is based on the following proposition that is proved in [Nad99]:

Proposition 2 *Let t be a de Bruijn term and let t' be a suspension term that yields t by annotation erasure. Further, suppose that the rules in Figure 1 allow t' to be rewritten to a head normal form in the generalized sense 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*

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

with a binder length of n as a head normal form in the conventional sense.

In the conventional setting, the reduction of a term to head normal form may be carried out by rewriting the head redex at each stage. Assuming the term is not already in head normal form, this redex is identified as follows: it is the term itself if it is a β-redex; otherwise, if the term is of the form $\lambda\, t$ or $(t\ s)$, thenx it is the head redex of t. In the suspension notation, there is one more possibility for the term and there is also a larger set of rewriting rules. The following definition takes these aspects into account and also anticipates graph based representations for lambda terms to generalize the notions of head redexes and head reduction sequences to its context.

Definition 3 *Let t be a suspension term that is not in head normal form.*

1. *Suppose that t has the form $(t_1\ t_2)_o$ or $(t_1\ t_2)_c$. If t_1 is an abstraction, then t is its sole head redex. Otherwise the head redexes of t are exactly the head redexes of t_1.*
2. *If t is of the form $(\lambda_c\, t_1)$ or $(\lambda_o\, t_1)$, its head redexes are identical to those of t_1.*
3. *If t is of the form $[\![t_1, ol, nl, e]\!]_c$ or $[\![t_1, ol, nl, e]\!]_o$, then its head redexes are t itself and all the head redexes of t_1.*

Let two subterms of a term be considered non-overlapping just in case neither is contained in the other. Then a head reduction sequence of a term t is a sequence $t = r_0, r_1, r_2, \ldots, r_n, \ldots$, in which, for $i \geq 0$, there is a term succeeding r_i if r_i is not in head normal form and, in this case, r_{i+1} is obtained from r_i by simultaneously rewriting a finite set of non-overlapping subterms that includes a head redex using the rule schemata in Figure 1. Obviously, such a sequence terminates if for some $m \geq 0$ it is the case that r_m is in head normal form.

Head reduction sequences may not be unique for two reasons: a term in any given sequence may have more than one head redex and there also may be choices in additional redexes to rewrite. However, this redundancy is inconsequential from the perspective of generating a head normal form and offers also a flexibility in reduction strategy that may be useful in practice:

Proposition 4 *A suspension term has a head normal form if and only if every head reduction sequence for it terminates.*

The essential idea in establishing this proposition, a detailed proof of which appears in [Nad99], is that of mapping head reduction sequences for suspension terms onto ones for the de Bruijn terms that correspond to them. A fully normalized form of a term may be generated by, as usual, reducing it first to a head normal form and then recursively applying this procedure to each of its arguments.

2.3 Variations on the Suspension Notation

The suspension notation that we have described here is actually a restricted version of the calculus presented in [NW98].[6] In particular, the full calculus allows for the transformation

[6] We are eliding annotations and rewrite rules that exploit these in this characterization.

of arbitrary nested suspensions as manifest in the expression $[\![[\![t, ol_1, nl_1, e_1]\!]_u, ol_2, nl_2, e_2]\!]_v$ into a *single* suspension of the form $[\![t, ol, nl, e]\!]_w$. The main task in this transformation is the computation of the effect of the substitutions embodied in the environment e_2 on each of the terms present in e_1. The richer calculus includes expression forms and rules that allow for this computation to be carried out through genuinely atomic steps. Now, there is a benefit to this kind of a merging ability: the transformation described permits all the substitutions to be effected in one walk over the structure of t rather than in the two walks that a naive processing of the nested suspensions would require. While the full collection of rules for merging environments offer considerable flexibility, this flexibility is also difficult to exploit in a practical reduction procedure. We have instead incorporated part of the power of the additional rules into the notation we use here in the form of two special rules: the (β_s') and the (r11) rules. These rules are, in fact, derived ones relative to the calculus in [NW98] and, as such, they collapse a larger sequence of 'merging' steps into single rule applications in two useful situations as we indicate next.

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 in Section 2.1, 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 can actually be understood as yielding in one step the final product of merging nested suspensions produced by two applications of the (β_s) rule. This rule in fact meshes well with a control regimen that follows a head reduction sequence and will be exploited in this fashion in the reduction procedures we describe. The rule (r11) is also redundant, but it serves a similar useful purpose in that it allows a reduction walk to be combined with a renumbering walk after a term has been substituted into a new (abstraction) context. In fact, the useful applications of rules (r11) and (r12) arise right after a use of rule (r5) and this behaviour can be made explicit through the following derived rules:

(r13) $[\![\#i, ol, nl, e]\!]_u \rightarrow t$,
provided $i \leq ol$, $e[i] = (t, l)$ and $nl = l$.

(r14) $[\![\#i, ol, nl, e]\!]_o \rightarrow [\![t, ol', nl' + nl - l, e']\!]_o$,
provided $i \leq ol$, $e[i] = ([\![t, ol', nl', e']\!]_o, l)$, and $nl \neq l$.

The reduction procedures that we present later will utilize these additional rules.

The annotations present in our terms allow a rapid simplification of suspensions in certain instances and they also facilitate a preservation of sharing in a graph based implementation of reduction. These capabilities are manifest in the rules (r8)-(r10) whose use we saw in our reduction example. To exploit these facets, it is necessary to introduce annotations into (de Bruijn) terms, something that we assume is done in a preprocessing phase. Our rewrite rules then conspire to preserve these annotations and to also utilize them where possible. It is possible, of course, to not utilize annotations in reduction. In this case, our terms can be simplified by dropping annotations from them altogether. Of course, the rules (r8)-(r10) will also have to be eliminated in this case and all the other rules will have to be changed to not mention annotations.

A final comment concerns the treatment of instantiatable or meta variables. The rule (r2) that is included in our collection for ‘reading’ such variables is based on a particular interpretation of them: 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 ‘graftable’ ones whose instantiation *can* contain free de Bruijn indices provides the basis for lifting higher-order unification to an explicit substitution notation [DHK00]. This is an interesting possibility but not one that we examine in this paper.

2.4 Relationship to Other Explicit Substitution Calculi

A variety of calculi have been proposed in recent years for reflecting substitution explicitly into term structure. From the perspective of practical deployment, these can be categorized based on whether or not they have machinery for combining substitutions arising from contracting different β-redexes into a single environment, thereby allowing them to be performed in the same traversal over a given term. The majority of the calculi that have been proposed do not, in fact, possess this ability. This is true, for example, of the $\lambda\upsilon$-calculus [BBLRD96], the $\lambda\zeta$-calculus [Muñ96], the λs_e-calculus [KR97] and the λ_{ws_o}-calculus [DG01]. The last two calculi are distinguished from the others in that they contain mechanisms for interchanging the order of substitutions in certain circumstances; such a device is needed to obtain confluence of rewriting under a graftable interpretation of meta variables. However, in the absence of such variables and when considering the implementation of reduction and comparison operations, all these calculi are similar to the suspension notation without the (β_s') rule.[7]

The systems that do permit the combination of reduction substitutions are the $\lambda\sigma$-calculus [ACCL91], the closely related ΛCCL calculus [Fie90] and the suspension notation. The first two calculi include machinery to combine *arbitrary* environments, in the same way that the full blown suspension notation that we mentioned in Section 2.3 does. However, this general ability affords too many choices and we believe that it also will need to be extracted into special derived rules such as the (β_s') rule before it can be embedded in actual reduction procedures. There is also a difference in the way in which these two variety of calculi encode adjustments that need to be made to substitution terms that appears to favour the suspension notation in practice. These adjustments are not maintained explicitly in the suspension notation 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

[7] The λs_e and λ_{ws_o} calculi both exploit certain optimizations that are possible when the environment contains exactly one nontrivial substitution. We take this aspect into account in our discussions in Section 6. The λ_{ws_o}-calculus permits the combination of some renumbering and substitution walks through its b_2 rule. Together with the λs_e-calculus, it also allows it to be recognized that, for instance, the substitution of t_2 cannot affect t_3 in the reduction of the term $((\lambda\,((\lambda\,(\lambda\,t_1))\;t_2))\;t_3)$ under any reduction strategy. However, neither calculus allows t_3 and (a suitably modified version of) t_2 to be substituted into t_1 in the *same* walk and this is the critical practical contribution of the (β_s') rule as later discussions highlight.

in the environment. Thus, consider a suspension term of the form $[\![t_1, 1, nl, (t_2, nl') :: nil]\!]_v$. 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 $[\![t_1, 1, nl, (t_2, (nl - nl')) :: nil]\!]_v$; 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 specific 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 $[\![(\lambda\, t_1), 1, nl, (t_2, (nl - nl')) :: nil]\!]_v$, we must produce one of the form $(\lambda_v\, [\![t_1, 2, nl + 1, @1 :: (t_2, nl - nl' + 1) :: nil]\!]_o)$. In contrast, using our notation, it is only necessary to add a 'dummy' element to the environment and to make a *global* change to the embedding levels of the overall term.

The above discussion, while correct in spirit, is inaccurate in one detail. None of the other calculi mentioned employ annotations in the way the suspension notation does and thus do not immediately afford the possibility of studying their utility in practice.[8] In summary, we believe that the suspension notation provides a concrete yet sufficiently general basis for examining the use of explicit substitution systems and the effect of the various choices afforded by them on actual implementation.

3 A Framework for Empirical Evaluation

We employ computations in the higher-order logic programming language λProlog [NM88] in this paper for the purpose of assessing the practical impact of the choices that exist in the representation of lambda terms and also in reduction strategies. There are two important reasons underlying this selection. First, λProlog is a language that genuinely employs lambda terms as data structures. In particular, it allows these terms to be used to represent complex syntactic objects whose structure involves binding and it includes mechanisms for manipulating such terms in logically meaningful ways. At a computational level, the use it makes of lambda terms is quite similar to what is done in logical frameworks, proof assistants and metalanguages such as Twelf [PS99], Isabelle [NPW02] and Coq [Tea02]. The observations that we make relative to this language therefore carry over naturally to all these other contexts. The second reason for our choice of experimentation platform is also quite compelling: we have access to a newly completed implementation of λProlog within which we can easily vary representation and reduction choices pertaining to lambda terms and then monitor the effects of these variations.

[8] The recognition and utilization of the property of independence of subterms from external abstractions is, however, part of two other systems. The Prolog/Mali implementation of λProlog [BR92] exploits such an idea and notes results relative to a specific program that are similar to the ones we observe in Section 7 [BR91]. The substitution erasure mechanism in [FMS04] also has an overlap with the way we use annotations.

In the rest of this section we outline the structure of the λProlog language briefly and then discuss the collection of programs over which we conduct our experiments as well as the kinds of data we collect to gauge performance.

3.1 The λProlog Language

The language λProlog is one that, from our current perspective, extends Prolog in three important ways. First, it replaces first-order terms—the data structures of a logic programming language—with the terms of a typed lambda calculus. Attendant on these lambda terms is a notion of equality given by the α-, β- and η-conversion rules. Second, λProlog uses the higher-order unification procedure [Hue75], which respects this extended notion of equality. Finally, the language extends the collection of goals or queries with two new kinds of expressions, these being of the form $\forall x\, G$ and $D \supset G$, in which G is a goal and D is a conjunction of clauses. These new goals, called *universal* and *implication* goals, respectively, have the following operational understanding: $\forall x\, G$ is solved by solving G with all free occurrences of x replaced by a new constant, and $D \supset G$ is solved by enhancing the existing program with the clauses in D and then attempting to solve G. Thus, at a programming level, universal goals provide a means for giving names a scope and implication goals similarly determine an extent for clauses that (partially) define procedures.

The new features in λProlog endow it with interesting metalanguage capabilities. To illustrate this facet, suppose that we wish to represent the terms of the pure, untyped lambda calculus with the intention of implementing reduction and other operations on them. For simplicity, assume that we deal only with closed terms. We designate a new atomic type *tm* to identify the class of objects encoding such terms in λProlog. Then, to capture the abstract syntax of an application, we introduce a new constant called *app* of type *tm*→*tm*→*tm*.[9] Thus, a lambda term of the form $(e_1\ e_2)$ will be represented by $(app\ \overline{e_1}\ \overline{e_2})$, where $\overline{e}$ denotes the representation of e. In representing abstractions, an interesting possibility arises. The binding content in such objects can be reflected into abstractions in the data structures of the metalanguage. Using this approach, referred to as the *higher-order abstract syntax* approach in [PE88], the lambda term $\lambda x\, e$ would be translated into the λProlog term $(abs\ \lambda x \overline{e})$, where *abs* is a special constructor with the type *(tm*→*tm)*→*tm* that is designated to identify encodings of abstractions.

The benefit of the 'higher-order' representation of abstraction is that many of the binding related operations on lambda terms become available directly because of the 'structural' understanding of abstraction embedded in λProlog. As a specific example, suppose that we desire to realize head normalization over (object-level) untyped lambda terms. The predicate *hnorm* implementing this operation can be defined through the following clauses:

```
hnorm X X :- bvar X.
hnorm (app A B) C :-
  whnorm A D, (D = (abs E), hnorm (E B) C ; C = (app D B)), !.
```

[9] There are devices in λProlog for identifying new types and constants with their associated types the details of which we do not go into out here.

```
hnorm (abs A) (abs B) :- ∀v (bvar v ⊃ hnorm (A v) (B v)).

whnorm X X :- bvar X.
whnorm (app A B) C :-
  whnorm A D, (D = (abs E), whnorm (E B) C ; C = (app D B)), !.
whnorm (abs A) (abs A).
```

The syntax used above follows that of Prolog with the exception that application is written in curried form, in keeping with the higher-order nature of the language. We note also the convention that application binds more tightly than **,** (the Prolog conjunction symbol) and `:-` (the reverse implication symbol). One significant aspect of this code is the treatment of substitution in contracting a β-redex. Thus, the substitution of B for the bound variable in the term represented by *(abs E)* is realized simply by applying E to B, allowing metalanguage reduction to do the rest. Another novelty is the realization of reduction within an abstraction context. This is handled, in essence, by introducing a new constant, annotating this as corresponding to a bound variable (via the *bvar* predicate), using this to temporarily dispense with the metalanguage abstraction and head normalizing the resulting structure. The scoping devices and λProlog reduction play a critical role in realizing this computation. After the head normalization of the body is realized, it is necessary to insert an abstraction over all occurrences of the introduced constant. This part of the computation is carried out by a restricted form of higher-order unification, manifest in the code by the matching of the produced structure with the term *(B v)*.

3.2 A Program Suite for Collecting Performance Data

The λProlog language has been used in a variety of applications such as implementing theorem provers, prototyping type inference systems and implementing compilers for programming languages. For our experimental study, we decided to categorize these programs based on the kind of computations they involve over lambda terms. We selected two representative programs from each category for collecting performance data. We explain our categorization and the selected programs in more detail below. One or two of our programs, such as the one involving computations over the Church encoding of natural numbers, do not represent significant applications for λProlog. However, they do strain aspects such as the reduction process and have also been used in other studies; for example, see [GL02].

Programs primarily requiring reduction: The lambda terms that are used in programs in this category figure mainly in reduction, the unification computation being largely first-order in nature. The two programs included in this category are the following:

SKI This program corresponds to an improved version of *hnorm*, the object-level head normalization procedure presented above, applied to arbitrary compositions of the well-known combinators *S*, *K* and *I*. The data that is collected is based on the *hnorm* procedure being applied to a collection of five hundred combinator compositions. We note that this program does use some higher-order unification, specifically of the ‘L_λ

variety,' but, in contrast with the next class of programs, most of the computation is through β-reduction.

Church This program involves arithmetic calculations based on Church's encoding of numerals and the combinators for addition and multiplication. The largest 'number' used in this program is around twenty thousand. The principal difference between this program and the previous one computationally is that the Church combinators are represented directly as lambda terms of third-order in the metalanguage, *i.e.*, the *app* and *abs* based encoding is not used. If the *app* and *abs* based encoding is employed, then only second-order computations are required at the meta-level regardless of the complexity of the object-level terms, leading to a restricted form of β-reduction being all that is used.

L$_\lambda$-style programs: This is by far the most important class from the perspective of λProlog applications. Computation in this class proceeds by first dispensing with *all* abstractions in lambda terms using new constants, then carrying out a first-order style analysis over the remaining structure and eventually abstracting out the new constants. As an idiom, this is a popular one amongst λProlog users and it has also been adopted in other related systems such as Elf and Isabelle. Programs in this class use a special case of reduction—the argument of a redex is always a constant—and a restricted form of higher-order unification [Mil91b]. We included the following programs in this category:

Compiler This is a compiler for a small imperative language with object-oriented features [Lia02]. Aspects of compilation had been studied in logical frameworks, but this relatively large experiment attempts to capture *all* the relevant stages in one setting. The program includes a bottom-up parser, a continuation passing-style intermediate language, and generation of native byte code. Significant parts of the computation in this program do not in fact involve lambda terms. However, there are also major parts that do and our study reveals that choices in representation of lambda terms and in reduction strategies here can have a significant impact on behaviour.

Typeinf A program that infers principal type schemes for ML-like programs [Lia97]. The representation of types treats quantification explicitly within this program and abstraction in the metalanguage is used to capture the binding effect. A type inference algorithm similar to that of [AS93] was used. Given the treatment of type variables, unification over types has to be explicitly programmed. Thus, many of the typical features of a metalanguage are exercised by this program. Another interesting aspect is that declarativity was taken seriously in developing the program and so (impure) control mechanisms, such as the cut familiar from Prolog, have been avoided.

Programs that require higher-order unification: The unification computation in the L$_\lambda$ class is carefully controlled: the process always terminates and unifiers are unique. These properties do not carry over to arbitrary higher-order unification. From a practical perspective, this means that there may be branching in unification. In particular, different lambda terms may have to be posited as bindings for instantiatable variables and reductions and other computations would have to be carried out, and possibly backtracked over, using such terms. The programs we included in this category are the following:

Hilbert This is an encoding in λProlog of the process of solving diophantine equations through higher-order unification [Mil92]. Solutions are not generated completely by this program in many instances. Rather, solvability is often determined, the exact identity of solutions being dependent on the unifiers for 'flexible-flexible' disagreement pairs left behind at the end of the computation. This program shares with *Church* the property of involving third-order terms that give rise to more involved reduction computations than seen in the L_λ case.

Funtrans This is a collection of transformations on functional programs [Mot00], such as through partial evaluation. Some elements of this program are similar to those in the *Compiler* example. However, this program tends to employ meta-level capabilities in a more direct manner, and includes third-order terms. The compiler, on the other hand, is restricted to second-order representations and retains greater control over the structure of terms, as is characteristic of the L_λ style of programming.

The web site at `www.cs.hofstra.edu/~cscccl/lambda-examples` contains the code for all the programs in our test suite and also has other information relevant to their use in our experiments.

3.3 Measuring Performance

All our experiments were conducted within the framework of the *Teyjus* system [NM99]. This system is an abstract machine and compiler based implementation of λProlog. Of special interest in the current context is the fact that *Teyjus* uses a low-level encoding of lambda terms based on the annotated suspension notation. Moreover, the use of the rewrite rules in Figure 1 are confined within it to a procedure that transforms terms into forms appropriate for subsequent comparison and unification computations. This procedure can be modified to realize, and hence to study, the effect of different choices in lambda term representation. It is this ability that we utilize in the following sections.[10]

A few comments on the parameters by which performance is measured are warranted. We present timing measurements in some of our test data. Where we do this, we obtained the data by running the relevant programs on a 400 MHz UltraSparc 5 system with 256 megabytes of memory and we report the average time over five runs. The timing data is useful in indicating trends and especially in understanding how significant the choices relating to reduction strategies and term representation are within a computational context that also involves unification, procedure invocation, backtracking and other related operations. However, we caution against putting too much weight on this factor in detailed

[10] In later sections we outline reduction strategies using Standard ML as a presentation vehicle. We note that this is really intended to be a 'pseudo-code' presentation: the actual procedures we use in the Teyjus system are all implemented in the C language.

assessments: while we have attempted to be as careful as possible to neutralize this effect, small variations in time may, for instance, be reflective of the absence or presence of specific optimizations in coding in the implementation language. We include two other measures in our tests relating to reduction strategies that we believe may be more accurate and, hence, more informative. One of these is the number of term nodes (*i.e.*, applications, abstractions, suspensions, atomic terms and 'cons' cells in environments) traversed during computation; this has an obvious and direct impact on time that is immune to the kinds of variations we mention earlier. The second is the number of new term nodes created on the heap.[11] This factor provides information about space requirements. We also note that the differences in heap usage statistics reflect principally new term nodes that are created and discarded during processing. Thus, although the *Teyjus* system currently does not have a garbage collection facility, give us also an indirect measure of the cost of garbage collection were such a mechanism to be included.

4 Choices in Reduction Strategies

The comparison of terms requires the production of head normal forms. There is flexibility, however, in the interpretation of these forms and their generation, at least some of which arises from the availability of explicit substitution calculi. Two specifically interesting choices may be understood as follows. First, it is possible to either utilize the encoding of substitution—*i.e.*, the suspension terms of the notation described in Section 2—only implicitly as an implementation level device or to reflect this also into actual term representation. Second, we may think of generating a fully normalized form for a term each time we need to examine its structure for comparison or we may stop after a head normal form has been exposed. At an intuitive level, these choices may be understood as those between an eager or a demand-driven approach to substitution and reduction, respectively.

We examine these choices more completely in this section and we assess also their consequences within a practical system. The suspension notation is used as the basis for this study. Towards ensuring that our observations apply uniformly to explicit substitution calculi, we avoid the use of annotations within this system for the moment; a consideration of the benefits of annotations will be taken up separately in a later section. However, we do assume rules for combining substitution walks—the (β_s') and (r11) rules relative to the suspension calculus—on the assumption that the style of processing they support would be standard within any environment based reduction procedure. We also assume a graph-based implementation of reduction, *i.e.*, lambda terms will be represented as graphs and destructive changes will be used to register, and thus to possibly share, reduction steps. The destructive treatment of reduction is orthogonal to that of variable binding for which we already assume the destructive realization with sharing that is customary in the logic programming setting.

[11] We implicitly consider only the term nodes seen or created by the reduction procedure in these counts since it is only these that are relevant to our study.

4.1 Eagerness versus Laziness in Substitution

The suspension notation provides a natural basis for extending usual environment based head normalization procedures to the situation where it is also necessary to look inside abstractions. Within such a procedure, suspensions will be generated to encode substitutions over terms and the reading rules will be used to calculate these out as needed. However, suspensions will not be represented explicitly. Rather, they will appear implicitly, in the form of the arguments of recursive calls to the normalization procedure; terms that are input to the procedure or that are eventually returned will not themselves contain suspensions. In the usual leftmost-outermost reduction control regime 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. These suspensions appear only at the top-level of terms and also will not persist beyond the reduction process.

A more detailed pseudo-code style presentation of the procedure outlined above is provided in Appendix A and we confine ourselves to a few qualitative remarks concerning its structure here. First, unlike usual reduction procedures, this one needs also to process the structures of terms embedded inside abstractions. Part of this requirement is manifest in the two extra components in a suspension in addition to a term and an environment, *i.e.*, the new and old embedding levels. These become arguments of the reduction procedure as well. Further, in descending into the bodies of abstractions, the (unannotated version of) rule (r7) will need to be used. However, for efficiency reasons, this rule should be utilized only when the external environment is non-vacuous and such care should be built into the normalization routine. Second, this procedure must return *de Bruijn* terms. Thus, after it has succeeded in exposing a generalized head normal form, it must enter a phase of processing in which any implicit suspensions over the arguments of this form are calculated out. This procedure exhibits, in this sense, a eager approach to substitution.

An alternative approach to that described above is to take the explicit substitution notation seriously by reflecting it directly into the structures of terms. The simplest way to realize this approach is to look at each point for a head redex in the term and to rewrite this explicitly and immediately using an appropriate rule from the collection in Figure 1. This processing structure can be easily translated into a recursive procedure that descends through terms looking for a leftmost-outermost redex to be rewritten next. However, such a procedure would adopt a somewhat naive approach to rewriting and would ignore the natural flow of control present in reduction. Thus, consider (the unannotated version of) 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 β-redex.

The procedure that uses suspensions only implicitly avoids the redundancy problem described above 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, leading to a different kind of redundancy: calculating out the substitutions over the arguments of a head normal form eagerly leads to a traversal of these arguments that is in addition to those that may be needed over these structures in the course of later processing.

Fortunately, it is possible to structure the reduction process so that both kinds of redundancies are avoided. The essential idea is to adopt the basic control regime of the environment based 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 a richer representation of terms that includes an encoding of suspensions. This raises the possibility that embedded suspensions may be encountered in the course of reduction and our new procedure must be equipped to deal with them. The notion of head reduction sequences described in Section 2.2 is, however, general enough to afford a simple way of dealing with this situation: the head normalization procedure itself can be invoked recursively on such embedded suspensions without the loss of any termination properties. This idea and other ones related to this style of processing are brought out in more detail in Appendix B through a pseudo-code presentation of the procedure under consideration.

The two procedures described above support different styles of effecting substitutions with contrasting benefits: eagerness can avoid redundant retracing of steps due to subsequent backtracking behaviour in a search-oriented framework whereas laziness has the potential for increasing sharing in structure traversal during substitution. Towards understanding the impact of these differences in practice, we collected data relating to time and space usage under each of these strategies for the six programs in our test suite. This information is presented in Table 1. The particular data that is tabulated here consists of the number of new term nodes created on the heap (referred to as *heap terms*), the number of term nodes encountered during normalization and substitution (referred to as *traversal count*) and the time required for the execution of each program (indicated in the columns labelled *time*). We note that the last component counts the time for the *entire* computation; in particular, it includes backchaining over clauses and unification computations in addition to normalization. We also remark that the traversal count includes only the number of times term objects are inspected to determine if it they are redexes or to percolate substitutions and that we do not include the creation of term nodes in this count.

The timing, structure traversal and structure creation measurements indicate a consistent advantage for the delayed substitution strategy. This advantage is at times dramatic. In the important cases of L_λ-style programs, structure traversal is reduced to about half the original (with an accompanying significant effect on overall processing time) and new structure creation is reduced to less than 25% by adopting this style of processing.

The better performance of the lazy substitution strategy is attributable, ultimately, to the fact that delaying creates substantially more opportunities for sharing in the structure

Program	*Eager Substitution*			*Lazy Substitution*		
	Heap Terms	*Traversal Count*	*Time (sec.)*	*Heap Terms*	*Traversal Count*	*Time (sec.)*
SKI	98,319	449,917	0.41	76,779	367,944	0.40
Church	44,797	175,301	0.18	37,162	169,192	0.17
Compiler	2,640,909	4,276,004	4.30	134,316	2,390,346	3.17
Typeinf	7,142,880	8,724,726	5.70	1,722,696	4,314,569	3.64
Hilbert	170,123	574,705	0.38	5,642	446,715	0.26
Funtrans	28,027	679,757	0.40	24,404	665,773	0.40

Table 1: Comparison of Substitution Strategies

traversal required for substitution and reduction. Some of these situations for sharing are created by the stalling of substitution walks till other computation steps have been carried out. As an illustration of this phenomenon, consider the manipulation of a quantified formula such as $\forall x \forall y\, P(x, y)$, where $P(x, y)$ itself represents a possibly complex formula containing occurrences of x and y. The encoding of this formula using lambda terms would take the form $(\mathit{all}\ \lambda x\, (\mathit{all}\ \lambda y\, \overline{P(x, y)}))$, where *all* is a constructor chosen to represent the universal quantifier and $\overline{P(x, y)}$ represents the encoding of $P(x, y)$. Now, consider a theorem-proving context in which a universal quantifier is processed by substitution with an instantiatable variable. In a language such as λProlog, this calculation would be effected by first recognizing a formula that fits the pattern $(\mathit{all}\ F)$ and then applying the instantiation of F to a new variable. When applied to the given formula, this kind of computation must produce the structure $\overline{P(X, Y)}$ where X and Y represent new instantiatable variables. In producing the structure, it is necessary to substitute X and Y for the bound variables x and y, respectively, in a common term. It is possible, in principle, to carry out both substitutions in one walk over this term. However, this effect can only be obtained if the actual effecting of substitutions can be delayed over theorem-proving steps (specifically, over the distinct calls to ∀-elimination). Such a behaviour is achievable under the lazy substitution strategy but *not* under the eager one.

Another situation in which a demand-driven approach to substitution can have beneficial effects is that when β-redexes appear embedded in the term into which substitution has to be performed. Suppose that this redex has the structure $((\lambda x\, t_1)\ t_2)$.[12] When an eager substitution strategy is used, the external substitution would first be percolated over this term, resulting in a walk over the structure of t_1. At a later point, the embedded redex may be contracted, producing another substitution traversal over t_1. If substitution is delayed until it is needed, these two distinct walks can actually be combined into one.

A remarkable fact about λProlog programs—one that we became aware of in trying

[12] We use a named notation for lambda terms here and below, contrary to our current assumption of the treatment of metalanguage expressions in an implementation, to ease the reading of these expressions by humans.

to understand the enormous performance differences seen above—is that structures that have significant quantities of embedded redexes can be produced whenever these programs embody an intrinsic use of higher-order unification. A central part of this computation is that of positing substitutions towards reconciling the differences between what are known as *flexible-rigid* disagreement pairs, *i.e.*, a pair of terms of the form

$$\langle \lambda x_1 \ldots \lambda x_l \, (F \; t_1 \; \ldots \; t_n), \lambda x_1 \ldots \lambda x_l \, (c \; s_1 \; \ldots \; s_m)\rangle$$

where F is an instantiatable variable, c is a constant or a bound variable occurrence captured by one of the abstractions in the binder of the term and $t_1, \ldots, t_n, s_1, \ldots, s_m$ are arbitrary terms; we assume here that the binder lengths of the two terms are identical, something that can be arranged based on typing considerations under the typical assumptions of equality between lambda terms. Using the procedure due to Huet [Hue75], a collection of substitutions known as the imitation and projection substitutions would be posited for F in this situation. These substitutions all have the structure

$$\{\langle F, \lambda w_1 \ldots \lambda w_n \, (h \; (H_1 \; w_1 \; \ldots \; w_n) \; \ldots \; (H_o \; w_1 \; \ldots \; w_n))\rangle\}$$

where h is either a constant or one of $w_1, \ldots, w_n$ and $H_1, \ldots, H_o$ are new instantiatable variables. Now, in subsequent steps of the computation, these new variables may themselves become instantiated in a similar way yielding embedded redexes at all the places where F appears. Moreover, the instantiations for the variables $H_1, \ldots, H_o$ may themselves contain embedded redexes, resulting in further embedded redexes in the binding for F.

A concrete understanding of the phenomenon described above may be obtained by considering a prototypical computation over a higher-order encoding of pure lambda terms described in Section 3. Assuming such a representation, the following collection of clauses in λProlog realize an object-evel equality predicate over lambda terms:

```
copy (app A B) (app C D) :- copy A C, copy B D.
copy (abs A) (abs B) :- ∀v(copy v v ⊃ copy (A v) (B v)).
```

These clauses embody an extremely popular style of programming in λProlog-like languages, seen already in Section 3, that uses universal and implication goals, substitution and higher-order unification to lift recursive computations over structures devoid of binding into ones that do include binding. Thus, consider a structure of the form $(abs\ \lambda x\, B)$ where B is an encoding only of applications. Assuming that we already possess the ability to make a copy of the latter kind of structure, we may proceed to make a copy of the former as follows. First, we replace all the occurrences of x in B with a chosen constant and make a copy of the resulting structure. We then abstract out all the occurrences of the constant from this copy and wrap an abstraction constructor around the generated object. The second clause above realizes just this kind of processing. We have chosen 'copying' as the task for simplicity here, but a variety of transformations can be effected on the given term using this

Number of Abstractions	*Number of Applications*	*Eager Substitution*		*Lazy Substitution*	
		Heap Terms	*Traversal Count*	*Heap Terms*	*Traversal Count*
3	3	429	986	169	684
3	12	2,670	4,097	520	2,070
12	3	5,880	6,865	1,094	2,956
12	12	20,629	20,896	2,983	7,162

Table 2: The Effect of Embedded Redexes

approach. Moreover, the *copy* predicate that is defined above can itself be used in different modes to realize operations such as unification and rewriting [Mil91a].[13]

Consider now the computation that will be engendered by a query of the form

$$\mathit{copy}\ (\mathit{abs}\ \lambda x\ (\mathit{abs}\ \ \lambda y\ (\ \mathit{app}\ x\ y)))\ F,$$

i.e., one that attempts to create in (the instantiatable variable) F a copy of the first argument supplied in the query above. By tracing the steps, we see that F will be bound to a term of the form *(abs B)*, then B will be bound to a term of the form $\lambda x\ (\mathit{abs}\ (H_1\ \ x))$, then H_1 will be bound to a term of the form $\lambda x\, \lambda y\ (\mathit{app}\ (H_2\ \ x\ \ y)\ (H_3\ \ x\ \ y))$ and, finally, H_2 and H_3 will be bound to the terms $\lambda x\, \lambda y\, x$ and $\lambda x\, \lambda y\, y$, respectively; note that the tokens beginning with uppercase letters all represent instantiatable variables here. The overall result of this computation is actually to produce in F a copy of the given term. However, the incremental manner in which this calculation is carried out leaves several embedded, uncontracted redexes in the binding determined for F. Piecing the various substitutions described above together, it can be seen that it is, in fact, the term

$$(\mathit{abs}\ \lambda x\ (\mathit{abs}\ ((\lambda x\, \lambda y\ (\mathit{app}\ ((\lambda x\, \lambda y\, x)\ \ x\ \ y)\ \ ((\lambda x\, \lambda y\, y)\ \ x\ \ y)))\ \ x)))$$

that F is bound to. An especially interesting point to note is that the embedded redexes all appear in what we might call ‘argument’ positions in this term. Thus, each of these redexes will be left in place by the head normalization procedure whenever it is invoked to manifest the top-level structure of a subterm of which the redex is a part.[14]

The computation that we have described above ends once a substitution for F has been determined. In this kind of situation the embedded redexes are a harmless artifact and do not significantly influence performance characteristics. However, in the typical situation it

[13] Meta-programming in a logic programming context sometimes require a *ground* representation in which object-level variables are represented as constants in the meta-language. The *copy* clauses can be used to bridge the resulting gap between meta- and object-level treatments of variables and substitution.

[14] An alert reader may wonder if the structure of the binding found for F is dependent on the particular higher-order unification procedure used. A little thought reveals that it is only the incremental manner in which the clauses lead to the generation of this binding that is the culprit. For example, F will be instantiated with an identical term even if the unification procedure described in [Mil91b], that also applies to this situation, is used.

is to be expected that bindings found for variables through ‘copy clause’ like processing are used in further computations. This kind of effect may be forced in the present context by invoking two *copy* goals in succession, *i.e.*, by invoking the goal

copy A B, copy B C

where A is a ground term represents the encoding of an actual lambda term and B and C are variables whose bindings are to be determined by the computation. Table 2 presents term traversal and structure creation data under the eager and lazy substitution strategies when we use for A the encodings of lambda terms consisting of a sequence of applications embedded within a sequence of abstractions but vary the number of such applications and abstractions. These data indicate a dramatic difference between the two styles for effecting substitutions, one that is especially exaggerated, as might be anticipated from the preceding discussion, as the number of abstractions at the head of the term increase. The performance differences noted relative to our test suite owe significantly to manifestations of this kind of phenomenon.

4.2 Eagerness versus Laziness in Reduction

In deeming that two terms are equal modulo the lambda conversion rules, it is necessary eventually to reduce them to their normal forms. As discussed in Section 2.2, head normalization provides the basis for a lazy approach to such reduction. In a situation where the check for equality also has a good chance of failing, such laziness can be advantageous: failure can be detected even without having to traverse the terms in question completely, let alone having to fully normalize them. However, we have also just seen some problems with this lazy strategy at least when it is utilized within a eager substitution regime. In particular, this approach can leave in place many embedded redexes even when effecting substitutions into them, requiring a subsequent redundant walk over their structure for the purpose of contracting them. Given the substantial impact on performance that such behaviour over embedded redexes has, a natural question to ask is whether an eager approach to reduction might not work better in practice. A further consideration in this regard is that if these redexes have to be contracted anyway, they are better contracted early and before a backtracking point is encountered so that rollbacks in computation do not cause such contractions to be undone and subsequently redone.

Towards understanding this issue, we conducted experiments comparing the head normalization strategy with one that fully normalizes terms anytime there is a need to look at their structure. In generating a fully normalized form of a term, we adopted both a call-by-value and a call-by-name or leftmost-outermost style of rewriting. The data are similar for both cases and we present only that observed under the latter strategy here.[15]

[15] Either style of rewriting can be legitimately used when strong normalizability holds for terms as it does in the context of interest. Of course, when only weak normalizability holds, the call-by-name approach must be used. Another observation, that is interesting especially in light of the work in [GL02], is that the call-by-value approach is compatible only with full normalization and cannot be used, for instance, in conjunction with the enhanced head normalization process we discuss next.

Program	*Head Normalization*			*Full Normalization*		
	Heap Terms	*Traversal Count*	*Time (sec.)*	*Heap Terms*	*Traversal Count*	*Time (sec.)*
SKI	76,779	367,944	0.40	98,319	1,535,208	0.76
Church	37,162	169,192	0.17	37,161	418,755,708	370.98
Compiler	134,316	2,390,346	3.17	150,625	32,317,036	12.68
Typeinf	1,722,696	4,314,569	3.64	1,629,028	10,590,519	4.87
Hilbert	5,642	446,715	0.26	13,142	1,736,602	0.68
Funtrans	24,404	665,773	0.40	20,414	1,273,737	0.53

Table 3: Comparison of Laziness and Eagerness in Reduction

In implementing the call-by-name approach, we essentially utilized the head normalization procedure that implicitly employs the suspension notation. However, once a generalized head normal form has been exposed, the same procedure is invoked on each of the arguments. Table 3 contrasts the heap usage, term traversal and running time observed under this interpretation of full normalization with those obtained when only head normalization is performed and a lazy approach to substitution is employed.

The data in Table 3 indicate a near parity in terms of the objects allocated on the heap between the lazy and the eager reduction strategies being considered. This actually reflects, as we had anticipated, a substantial reduction in the creation of new term structures if a full normalization strategy is used instead of a head reduction approach that also carries out substitution eagerly. However, the present form of full normalization is clearly not practically viable. The decision to completely normalize a term each time there is a need to look at it can lead to several traversals over the structure of any given term. Moreover, *all* these traversals are redundant if the term is already in normal form. This intuition is borne out by the differences in structure traversal (and processing time) between the two reduction approaches tabulated in Table 3. The difference is most dramatic in the case of the *Church* program where it arises out of the need to repeatedly traverse large terms encoding numerals. Even if this example is skewed towards revealing the shortcomings of the full normalization approach, the point remains that there *are* circumstances under which performance can deteriorate in an unacceptable way under it.

A closer consideration of the data presented to this point indicates a reduction approach that is intermediate between head normalization and full normalization that may be useful to try. This approach would be driven by the needs of the comparison process and its first goal would be to produce a head normal form. Once such a form has been exposed, it would proceed to effect substitutions eagerly in the argument parts of the form, rather than leaving these to be completed later as suspensions. However, in contrast to the head normalization with eager substitution approach, the current procedure would also contract β-redexes it encounters in the course of any structure traversal it undertakes. In other words, unlike full normalization, this procedure will contract non-head redexes only if it encounters these in

Program	*Lazy Substitution*			*Enhanced Head Normalization*		
	Heap Terms	*Traversal Count*	*Time (sec.)*	*Heap Terms*	*Traversal Count*	*Time (sec.)*
SKI	76,779	367,944	0.40	98,319	459,198	0.40
Church	37,162	169,192	0.17	37,161	213,095	0.22
Compiler	134,316	2,390,346	3.17	109,589	2,452,932	3.13
Typeinf	1,722,696	4,314,569	3.64	1,588,648	4,936,116	3.49
Hilbert	5,642	446,715	0.26	12,908	461,851	0.27
Funtrans	24,404	665,773	0.40	19,734	676,067	0.41

Table 4: Performance Under an Enhanced Form of Head Normalization

the course of performing substitutions.

The procedure that is outlined above can be obtained by a straightforward modification to that in Appendix A; essentially the call to the substitution process in this procedure needs to become a recursive call to the normalization process instead. We, once again, experimented with its use within the *Teyjus* system. Referring to the new approach as *enhanced* head normalization, Table 4 contrasts performance under it to that under the head normalization with lazy substitution regime. The new procedure is designed to avoid the multiple walks over the bodies of embedded redexes that affected adversely the head normalization with eager substitution approach as well as the redundant walks over already normalized structures that mark the full normalization scheme. The data bear out the fact that this effect is, indeed, achieved.

The enhanced head normalization procedure still has a disadvantage in principle in comparison with the lazy substitution approach in that it is not possible to combine structure traversals arising from contracting redexes whose generation is dependent on other computation steps. It is therefore somewhat surprising that this procedure out-performs the lazy substitution approach in some of the test programs considered. Our understanding of this phenomenon is as follows: There are certain places in the evaluation of queries where it is necessary to fully normalize terms. In these circumstances, the benefit of the kind of sharing described for the lazy substitution approach is lost. Furthermore, eagerness in substitution and reduction can be also be an advantage in this context since if these computations are performed *before* backtrack points, then they will not have to be rolled back and subsequently redone.

We illustrate the situation described above through a suitable enhancement of the *copy* clauses based example. Suppose first that we add to our language of pure lambda terms a *let* construct that takes the form

$$let\ x = e\ in\ t\ end.$$

In this construct, whose semantics we assume is familiar to the reader, x is expected to be a variable and e and t can be arbitrary terms. The higher-order encoding of this construct will take the form

$(let\ \overline{e}\ \overline{\lambda x\, t})$

where *let* is a special constructor of type *tm*→*(tm*→*tm)*→*tm* and $\overline{e}$ and $\overline{\lambda x\, t}$ are encodings of e and $\lambda x\, t$, respectively. Now, suppose we want to extend our *copy* predicate to the language that includes the *let* construct with the following additional possibility: in the case that the variable x bound in the construct does not appear in the body, then the overall construct is to be considered equal to just the body. This desire can be realized through the following, augmented definition of the *copy* predicate:

copy (app A B) (app C D) :- *copy A C, copy B D.*
copy (abs A) (abs B) :- $\forall v$*(copy v v* ⊃ *copy (A v) (B v)).*
copy (let E (λx T)) T' :- *copy T T'.*
copy (let E T) (let E' T') :- *copy E E',* $\forall v$ *(copy v v* ⊃ *copy (T v) (T' v)).*

There are two clauses for the *let* form of expressions in this definition, the second one of which is easy to understand by analogy with the clause for abstractions. The first of these clauses, on the other hand serves to eliminate vacuous *let*s. The key part of this clause is the term $\lambda x\ T$ that is used in the pattern to be matched with the first argument of the *copy* predicate. This term will unify only with an abstraction structure whose body *does not* contain an occurrence of the bound variable. The *let* is vacuous in this case and the clause allows for its elimination in the second, 'copy', argument.

Let us consider now the task of solving a query of the form

copy (abs λx (let t_1 t_2*)) C*

in which t_1 and t_2 represent arbitrary terms and C is an instantiatable variable. Processing the top-level abstraction structure will produce a redex. This redex will be contracted and completely evaluated prior to the *let* structure being examined in the case that an eager approach to substitution and reduction is used. In contrast, the actual performance of substitution will be carried out only incrementally under the lazy substitution scheme. However, the pattern in the first clause pertaining to *let* requires a sort of 'occurs-check' to be performed and the incoming term will have to be fully normalized in the course of carrying out this check. Suppose now that the *let* under consideration is actually not a vacuous one. In this case, the occurs-check will fail at some point, all the reduction and substitution work carried out at the behest of this clause will be rolled back, only to be repeated in the course of using the second clause for the *let* case. In short, delaying substitutions when coupled with backtracking can lead to a redundancy in structure traversal that is not also present under the eager approach. It is precisely this kind of phenomenon that gives the enhanced form of head normalization the edge, even if ever so slightly, in some of the examples included in the test suite.

5 The Treatment of Bound Variables

We have assumed up to this point the de Bruijn scheme for treating bound variables. There is another method that is commonly used both in discourse and in implementation, this being that of representing bound variables with explicit names or, roughly equivalently, by pointers to cells associated with their binding occurrences. In this section we contrast these two approaches. In making this comparison, we note at the outset the importance of distinguishing two operations on which the treatment of bound variables has an impact. One of these is that of checking the identity of two terms up to α-convertibility. The second is that of making substitutions generated by β-contractions into terms, in which case the treatment of bound variables is relevant to the way in which illegal capture is avoided.

The de Bruijn representation is evidently superior from the perspective of checking the identity of terms up to α-convertibility. For example, consider matching the two terms $\lambda y_1 \ldots \lambda y_n\,(y_i\ t_1\ \ldots\ t_m)$ and $\lambda z_1 \ldots \lambda z_n\,(z_i\ s_1\ \ldots\ s_m)$. The heads of these terms that are embedded under the abstractions are bound in both cases by the same abstraction. Thus, the matching problem can be translated into one over the arguments of this term. A prelude to this transformation at a formal level under a name based scheme is, however, a 'normalization' of bound variable names. This step is avoided under the de Bruijn scheme.

A further consideration of the above example indicates a more significant advantage of the de Bruijn notation. Under a name based scheme, the transformation step must produce the following set of pairs of terms to be matched:

$$\{\langle \lambda y_1 \ldots \lambda y_n\, t_1, \lambda z_1 \ldots \lambda z_n\, s_1\rangle, \ldots, \langle \lambda y_1 \ldots \lambda y_n\, t_m, \lambda z_1 \ldots \lambda z_n\, s_m\rangle\}.$$

The abstractions at the front of each of the terms are necessary: they provide the context in which the bound variables in the arguments are to be interpreted in the course of matching them. Constructing these new terms at run time is computationally costly and also a bit too complex to accommodate in a low-level, abstract machine based system such as *Teyjus*. A possibility in the named-based scheme that avoids the specific problem of carrying forward the abstractions is to first apply the two terms being compared to a sequence of distinct, new variables [GL02] or constants [BR92]. This approach has the disadvantage at least of requiring an additional substitution into terms and also introduces symbols into terms whose origins or meanings are difficult to comprehend at the source language level. The latter factor can be significant especially when the comparison operation is distributed over other computation steps and the results have to be occasionally shown back to the user. Under the de Bruijn scheme, the abstraction context is implicitly present in the numbering of bound variables, obviating the explicit attachment of the abstractions and, moreover, the index of the variable occurrence conveys its significance in an unequivocal way.

From the perspective of carrying out substitutions in contrast, the de Bruijn scheme has no real benefit and may, in fact, even incur an overhead. The important observation here is that the renaming that may be needed in the substitution process in a name based scheme has a counterpart in the form of renumbering relative to the de Bruijn notation. To understand the nature of the needed mechanism, we may consider the reduction of the term $\lambda x\,((\lambda y\,\lambda z\,y\ x)\ (\lambda w\,x))$ whose de Bruijn representation is $\lambda\,((\lambda\,\lambda\,\#2\ \#3)\ (\lambda\,\#2))$. This term reduces to $\lambda x\,\lambda z\,((\lambda w\,x)\ x)$, a term whose de Bruijn representation is $\lambda\,\lambda\,((\lambda\,\#3)\ \#2)$. Comparing the two de Bruijn terms, we notice the following: When substituting the term $(\lambda\,\#2)$ inside an abstraction, the index representing the locally free variable occurrence,

Program	*Lazy Substitution*		*Enhanced Head Normalization*	
	Total Substitutions	*Renumbering Substitutions*	*Total Substitutions*	*Renumbering Substitutions*
SKI	16,749	643	16,749	1005
Church	35,824	200	35,824	200
Compiler	93,766	0	60,677	0
Typeinf	780,745	0	656,950	0
Hilbert	1,718	412	5,977	1,773
Funtrans	8,367	96	7,866	96

Table 5: The Frequency of Renumbering with the de Bruijn Representation

i.e., 2, has to be incremented by 1 to avoid its inadvertent capture. Further, indices for bound variable occurrences within the scope of an abstraction that disappears on account of a β-contraction may have to be changed; here the index 3 corresponding to the variable occurrence x in the scope of the abstraction that is eliminated must be decremented by 1. The substitution operation that is used in formalizing β-contraction under the de Bruijn scheme must account for both effects.

At a detailed level, there is a difference in the renaming and renumbering devices needed in name-based and nameless representations. Given a β-redex of the form $(\lambda x\, \lambda y\, t_1)\ t_2$ whose de Bruijn version is a term of the form $(\lambda\, \lambda\, \hat{t}_1)\ \hat{t}_2$, the renaming in the first case is effected over the 'body', *i.e.*, $\lambda y\, t_1$, and in the second case over the argument, *i.e.*, $\hat{t}_2$.[16] One advantage of the name-based representation is that the renaming may be avoided altogether if there is no name clash. However determining this requires either a traversal of the term being substituted, or an explicit record of the variables that are free in it. An interesting alternative, described, for instance, in [AP81], is to always perform a renaming and, more significantly, to fold this into the structure traversal that realizes the β-contraction substitution.

The above discussion indicates that the additional cost relative to substitution that is attendant on the de Bruijn notation is bounded by the effort expended in renumbering substituted terms. A first sense of this cost can thus be obtained by measuring the proportion of substitutions that actually lead to nontrivial renumbering of compound terms. Cases of this kind can be identified as those in which rule (r5) is used where the skeletal term is non-atomic and an immediate simplification by rule (r12) is not possible.

Table 5 tabulates the data gathered towards this end for the two viable and comparable approaches that was identified by the analysis in Section 4, namely head normalization with lazy substitution and the enhanced form of eager head normalization. An interesting observation is that *no* renumbering is actually involved in the case of L_λ style programming. The reason for this is that the only reductions performed are those corresponding to

[16] In the de Bruijn scheme, some bound variables in $\hat{t}_1$ may also have to be renumbered, but this can be done efficiently at the same time that $\hat{t}_2$ is substituted into the term.

Program	*Lazy Substitution*		*Enhanced Head Normalization*	
	Separate Renumbering	*Merged Renumbering*	*Separate Renumbering*	*Merged Renumbering*
SKI	23,884	23,884	23,884	23,884
Church	13,063	13,063	13,063	13,063
Compiler	102,592	102,592	70,062	70,062
Typeinf	1,441,489	1,441,489	1,314,324	1,319,901
Hilbert	3,188	3,188	4,860	4,860
Funtrans	9,508	9,508	9,422	9,462

Figure 2: The Effect of Separating Contraction and Renumbering Walks

eliminating the binding with a new constant. Thus, for a significant set of computations carried out in λProlog and related languages, renumbering is a non-issue. In the other cases, some renumbering can occur. A point worth noting is that an enhanced ability to combine substitutions, as is manifest in the head normalization procedure that carries out substitutions lazily, appears to reduce renumbering work significantly. This phenomenon is also understandable; substituting a term in after more enclosing abstractions have disappeared due to contractions leaves fewer reasons to renumber.

The cases where a nontrivial renumbering needs to be done do not by themselves constitute an extra cost. In general, when a term is substituted in, it is necessary also to examine its structure and possibly to reduce it to (weak) head normal form. Now, the necessary renumbering can be incorporated into the same walk as the one that carries out this introspection. This structure is realized by choosing to percolate the substitution inwards first in a term of the form $[\![t, 0, nl, nil]\!]_v$, using rule (r11) to facilitate the necessary merging in the case that t is itself a suspension. The main drawback of this approach, in contrast to the scheme in [AP81] for instance, is that it can lead to a loss in sharing in reduction if the same term, t, has to be substituted, and reduced, in more than one place. The alternative strategy would separate the contraction and renumbering phases by contracting t first. An indication of the loss in sharing can be obtained from the differences in the number of β-redexes encountered under the two strategies. Table 2 tabulates the data relevant to this assessment, again under the lazy enhanced head normalization schemes. There is actual loss in sharing in β-contractions in the case of only one program from our test suite and even in this case the loss is a very small fraction of all the redexes contracted.

The conclusion from these data and discussions seems to be that the de Bruijn treatment of bound variables is the preferred one in practice. It is *obviously* superior to name based schemes relative to comparing terms modulo α-conversion; in fact, for reasons that we have explained, we know of no way to use a name based representation in a low-level implementation that is still user-friendly. The de Bruijn scheme has a potential drawback in terms of a renumbering overhead in realizing β-contraction. However, our experiments show that this overhead is either negligible or nonexistent in most cases.

Program	*With Merging*			*Without Merging*		
	Heap Terms	*Traversal Count*	*Time (sec.)*	*Heap Terms*	*Traversal Count*	*Time (sec.)*
SKI						
lazy	76,779	367,944	0.40	135,691	431,750	0.46
enhanced	98,319	459,198	0.40	137,720	504,125	0.46
Church						
lazy	37,162	169,192	0.17	627,048	782,972	0.58
enhanced	37,161	213,095	0.22	626,582	826,124	0.65
Compiler						
lazy	134,316	2,390,346	3.17	2,417,025	5,226,479	5.03
enhanced	109,589	2,452,932	3.13	1,475,457	4,018,991	4.20
Typeinf						
lazy	1,722,696	4,314,569	3.64	15,695,839	18,936,333	13.06
enhanced	1,588,648	4,936,116	3.49	13,072,813	16,484,357	11.05
Hilbert						
lazy	5,642	446,715	0.26	25,074	464,505	0.28
enhanced	12,908	461,851	0.27	58,565	503,727	0.30
Funtrans						
lazy	24,404	665,773	0.40	55,290	701,188	0.45
enhanced	19,734	676,067	0.41	48,548	708,866	0.43

Table 6: Impact of Merging Rules

6 The Value of Mechanisms for Combining Substitutions

The suspension notation includes a rule that enables the collection of multiple reduction substitutions into one environment, this being the (β_s') rule. However, not all explicit substitution calculi have this kind of combining ability. On the one hand, omitting such an ability has potential theoretical benefits: it is known at least in some cases to lead to the preservation of strong normalization properties of the underlying lambda calculus [BBLRD96, DG01]. On the other hand, we believe that an approach that does not use a mechanism for collapsing multiple substitution traversals into a composite one may not be viable in practice. The environment based reduction procedures that we considered in Section 4 have all utilized the (β_s') rule whenever possible for this reason. Given that the ability to structure reduction in this way distinguishes between explicit substitution calculi, it is important to quantify this observation.

It is possible to simulate the effect of not using the ability to combine reduction substitutions in our setting by simply choosing to employ the (β_s) rule to contract *every* β-redex. Table 6 presents data indicating the impact of the combination ability using this idea. In particular, it tabulates the number of term nodes created on the heap, the term traversal

Program	*Lazy Substitution*		*Enhanced Head Norm*	
	With Merging	*Without Merging*	*With Merging*	*Without Merging*
SKI	153,108	153,998	188,544	171,982
Church	231,531	698,978	253,486	720,789
Compiler	966,331	1,293,564	602,730	967,120
Typeinf	6,282,150	12,533,011	5,564,191	10,647,547
Hilbert	19,088	21,214	46,427	65,618
Funtrans	43,043	49,263	42,545	47,988

Table 7: Variable Lookup Costs With and Without Substitution Combination

count, and the composite running time for the programs in our test suite in the cases when the (β_s') rule was and was not used for the head normalization procedure with delayed substitution (denoted by *lazy*) and the enhanced form of the head normalization procedure with eager substitution (denoted by *enhanced*). We note that the rule (r11) also represents a form of substitution combination that is missing in some of the calculi of explicit substitution. However, since some form of this rule is available in other calculi, we allowed it to be employed in our reduction procedures. The data indicate a clear and significant superiority for both reduction strategies when the combination ability is also exploited. That the source of this difference is, in fact, the use of the (β_s') rule is also not difficult to substantiate. We have compared the number of times the (β_s') rule is used to contract a β-redex with the total number of β-redexes contracted. This rule turns out to be used in well over 50% of all the cases. In the case of the L_λ-style programs, the usage is actually close to 90% under the lazy substitution strategy and also in the enhanced form of head normalization.

In the discussion above, we have ostensibly overlooked an optimization that is possible in the situation where the underlying explicit substitution calculus does not include the possibility of encoding multiple non-trivial substitutions in a single environment. In such a situation, it is not necessary to represent environments at all. Rather, it suffices to simply encode the (single) term to be substituted and the index of the variable for which it is to be substituted; the actual substitution and all the adjustments to indices can be calculated from just this information. There is an impact to this optimization along both the space and time dimension, assuming a list based representation for environments. The data in Table 6 actually already factors in the space optimization. In particular, in counting the number of new term nodes, we have not included the cost of creating environment terms in the case when the merging ability available through the (β_s') is not utilized. With regard to timing considerations, in the case that an environment represents multiple substitutions, there is a cost associated with looking up the binding corresponding to a particular index as required by rules (r4) and (r5); this cost arises from having to work down the list until the right entry has been found. This lookup cost is unitary when the 'environment' encodes a single substitution. However, in this case, in contrast to the situation where environments are capable of encoding more than one substitution, *several* lookups may have to be done on the same variable. There appears thus to be a tradeoff between the two approaches with regard to this aspect. Table 7 contrasts this lookup cost for the two reduction strategies under consideration, assuming a unit cost for proceeding to the next item in an environment list and for effecting a substitution once the item corresponding to the index has been found. As we see from this data, the situation where multiple substitutions can be merged into one environment is no worse with regard to this cost than that in which they cannot be so merged, and is actually noticeably better in some instances.

7 The Relevance of Annotations

We have not used the annotation scheme supported by the suspension notation in our normalization procedures up to this point. As we have seen in Section 2, these annotations can facilitate a quick calculation of substitution in some cases and they can also lead to greater sharing in the rewriting process. There is, of course, a cost associated with maintaining and utilizing annotations. However, this cost can be considerably reduced with proper care. In the *Teyjus* implementation, for example, an otherwise unused low-end bit in the tag word corresponding to each term node stores this information. The setting of this bit is generally folded into the setting of the entire tag word and a single test on the bit independently of the type of the term node suffices in utilizing the information in the annotation.

There is actually a situation discussed in Section 4 in which annotations have a significant potential to be useful, this being that of embedded (β-)redexes in terms. Recall that these kinds of redexes arose when a term of the form

$$\lambda w_1 \ldots \lambda w_n \, (h \ (H_1 \ w_1 \ \ldots \ w_n) \ \ldots \ (H_m \ w_1 \ \ldots \ w_n))$$

is posited as a substitution for a variable F, with the variables $H_1, \ldots, H_m$ becoming bound through later computations to abstraction terms. The problem posed by such embedded redexes is that if they are not contracted at the time of effecting the substitution generated by contracting the application of F, then a separate, redundant walk would have to be performed later over the bodies of their ‘function’ parts when they are themselves contracted. The interesting thing to note, however, is that these parts must actually be closed terms and can be annotated as such at the point when substitutions are created for the variables $H_1, \ldots, H_m$. In the presence of these annotations, the contraction of the redex corresponding to the application of F to its arguments generates *no* traversals over the structures that instantiate $H_1, \ldots, H_m$ and the problematic redundancy in substitution walks can be avoided in this way.

An experiment was conducted to test out this potential benefit from annotations. The data from this experiment is tabulated in Table 8. For each of the programs in our test suite, we show in two rows the number of new term nodes allocated on the heap and the number of term nodes encountered during substitution and normalization, first when annotations were used and then when annotations were not used. Moreover, this data is presented for each of

Program	*Eager Substitution*		*Lazy Substitution*		*Enhanced Head Norm*	
	Heap Terms	*Traversal Count*	*Heap Terms*	*Traversal Count*	*Heap Terms*	*Traversal Count*
SKI	45,884	391,004	46,362	360,327	45,884	402,271
	98,319	449,917	76,779	367,944	98,319	459,198
Church	37,596	171,133	37,162	169,192	37,161	213,095
	44,797	175,301	37,162	169,192	37,161	213,095
Compiler	133,730	2,501,384	134,316	2,390,346	109,447	2,452,690
	2,640,909	4,276,004	134,316	2,390,346	109,589	2,452,932
Typeinf	1,654,245	4,808,113	1,722,694	4,314,569	1,588,576	4,936,080
	7,142,880	8,724,726	1,722,696	4,314,569	1,588,648	4,936,116
Hilbert	169,748	574,419	5,642	446,715	12,908	461,851
	170,123	574,705	5,642	446,715	12,908	461,851
Funtrans	19,165	671,751	23,762	665,098	18,749	674,751
	28,027	679,757	24,404	665,773	19,734	676,067

Table 8: The Effect of Annotations

the reduction procedures studied in Section 4. As expected from the earlier discussion, the reduction strategy that shows the most improvement from the use of annotations is the one that suffers the most from the presence of embedded redexes, namely the head normalization strategy with eager substitution. The improvement in this case is, in fact, quite dramatic: with annotations, performance under this strategy becomes comparable to that under the others in all cases except that of the *Hilbert* program. Annotations provide very little benefit along the dimensions measured with the other strategies. This is also not surprising. In the cases of both head normalization with eager substitution and the enhanced form of head normalization, most of the traversals over a given structure are combined into one. Since at least one walk has to be carried out over a term in any practically interesting situation, annotations do not help in avoiding any of this work. Moreover, since almost all of these 'accumulated' substitutions are likely not to be trivial in the sense of being effected over a closed term, annotations do not also lead to any increased possibilities for sharing in reduction either.

The observations in the case of the *Hilbert* program are an exception to the above generalizations. From a closer analysis of the terms encountered in the course of execution of this program, we understand that embedded redexes are the key to the differences in behaviour under the varied strategies here as well. However, these embedded redexes arise from a different phenomenon, namely an interaction between the presence of third and higher order terms and the conversion of terms to an η-expanded form that is done to account for equality under the η-conversion rule. As a specific example, consider the term

$$\lambda u\,((\lambda x\,\lambda y\,\lambda z\,(y\ (x\ z)))\ u)$$

in which the bound variable u has a function type with its argument also being of function type. (We use a named lambda calculus here for readability.) Now, under an η-expansion, this term becomes

$$\lambda u\,((\lambda x\,\lambda y\,\lambda z\,(y\ (x\ z)))\ \lambda w\,(u\ \lambda v\,(w\ v))).$$

Notice that the the subterm $\lambda w\,(u\ \lambda v\,(w\ v))$ is an *open* term and any term that it is applied to will consequently also be open. The contraction of the outermost redex in the displayed term then produces an embedded redex whose function part must be annotated as open. Many embedded redexes in the *Hilbert* program are, in fact, generated this way and annotations do not provide assistance to the eager substitution strategy towards avoiding redundant traversal over the function part of these redexes for this reason.[17]

8 Conclusion

This paper has examined a collection of issues relevant to the representation of lambda terms and the realization of reduction in the situation where these terms are used as data structures, thereby requiring their intensions to be taken seriously. Explicit substitution calculi provide an excellent basis for the machine encoding of these terms in such contexts but, as we have noted, detailed decisions on the actual deployment of such calculi require a careful understanding of the computations that arise in practice. We have tried to obtain such an understanding by experimenting with different ways of using the suspension notation in an implementation of the λProlog language and by measuring the effects of such variations on the performance of prototypical programs. The framework for our experiments has several characteristics that lends generality to our observations: the suspension notation contains within it the spectrum of mechanisms that are found in explicit substitution calculi and the computations that are carried out within the λProlog language typifies those that are performed within systems in which lambda terms are used for representing objects.

The conclusions of our study are varied and may be summarized as follows. First, the de Bruijn scheme appears to be an important one to use in representing lambda terms. The overheads in reduction because of this scheme are minimal and are offset by significant benefits in the encoding of binding contexts and in the checking of the identity of terms modulo the renaming of bound variables; we have, in fact, not been able to conceive of a viable low-level implementation that uses a name based representation for reasons that we have discussed in Section 5. Second, the ability to combine substitutions generated by the contraction of β-redexes is an important one and explicit substitution calculi that cannot support this possibility are not viable in practice. Third, there are a number of somewhat unanticipated factors that affect the choice of an 'optimal' reduction strategy for terms. These include the fact that the incremental instantiation of variables during computation can give rise to a large number of embedded β-redexes whose contraction should be carefully coordinated and that the backtracking needed to support a search-based paradigm

[17] The alert reader may wonder why the *Church* program that also involves third-order terms avoids the fate of *Hilbert*. It does this by exploiting polymorphic typing in a way idiocyncratic to the *Teyjus* system. We eschew a detailed discussion of this matter since it is orthogonal to our present scope.

can sometimes favour an eager performance of substitution and reduction work. Fourth, notwithstanding the last observation, a strategy that attempts to fully normalize terms each time their structure is to be examined is not a viable one in practice. Fifth, an enhanced head normalization strategy that performs substitutions eagerly but also considers contracting β-redexes that are encountered during substitution traversals turns out, surprisingly, to be competitive with a head normalization strategy that delays actual substitution over argument structures. Finally, while head normalization under an eager substitution regimen is not practical by itself, it can be made competitive with the previous two approaches by using annotations on terms to indicate their dependence on external abstractions.

The work reported here can be extended in several ways. One aspect that is open to further investigation is the compiled realization of reduction. Recent work relative to the *Coq* system has shown how to do this assuming eager reduction and substitution strategies to obtain substantial speedups in comparison with the existing interpretive approach [GL02]. Now, we have seen that full normalization is not a sensible strategy in a situation where the structures of terms are to be examined incrementally and, indeed, the examples considered in the mentioned study seem to be ones where the comparison is limited to ground terms available in complete form at the beginning of the computation. Nevertheless, compilation does have benefits and it is of interest to see if these can be harnessed to yield better implementations of head normalization with delayed substitutions or the enhanced form of head normalization. Another matter that is useful to consider is the difference between destructive and non-destructive realizations of reduction. There are ‘obvious’ advantages to a destructive version in a deterministic setting that become less clear with a language like λProlog that permits backtracking. This matter can be examined experimentally. The final matter that we mention concerns the implementation of higher-order unification. It has been shown in [DHK00] how to lift this operation to an explicit substitution notation with a special, graftable interpretation of meta variables. Doing so has the benefit of making the application of substitutions to such variables very efficient. However, there are also costs: a more general, and complete, mechanism for combining substitutions is needed and additional information must be also be retained dynamically. There is a tradeoff here that can, once again, be assessed empirically.

9 Acknowledgements

We are grateful to the reviewers of this paper for their careful reading and thoughtful feedback on this paper. This work has derived supported from a special research grant from the Department of Computer Science at Hofstra University, NSF Grant CCR-0096322, a Grant in Aid of Research from the University of Minnesota and the Department of Computer Science and Engineering and the Digital Technology Center at the University of Minnesota. Revisions were made to this paper during Nadathur’s sabbatical visit to the Parsifal and Comete groups at Ecole Polytechnique and INRIA, Saclay.

References

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AP81] L. Aiello and G. Prini. An efficient interpreter for the lambda-calculus. *The Journal of Computer and System Sciences*, 23:383–425, 1981.

[AS93] A. and Z. Shao. Smartest Recompilation. In *Tenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1993. Longer version as Princeton University Technical Report CS-TR-395-92.

[BBLRD96] 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.

[BR91] Pascal Brisset and Olivier Ridoux. Naive reverse can be linear. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 857–870. MIT Press, June 1991.

[BR92] Pascal Brisset and Olivier Ridoux. The compilation of λProlog and its execution with MALI. Publication Interne 687, IRISA, Rennes, November 1992.

[Bru72] 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.

[Bru80] N. de 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.

[CAB$^+$86] 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.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[DG01] R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structures for Computer Science*, 11:169–206, 2001.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.

[Fie90] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–15. ACM Press, January 1990.

[FMS04] M. Fernandez, I. Mackie, and F-R. Sinot. Closed reduction: Explicit substitutions without alpha-conversion. *Mathematical Structures in Computer Science*, 2004. To appear.

[GL02] 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.

[Har86] Robert 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.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[KR97] Fairouz Kamareddine and Alejandro Ŕios. 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.

[Lia97] C. Liang. Let-polymorphism and eager type schemes. In *TAPSOFT '97: Theory and Practice of Software Development*, pages 490–501. Springer Verlag LNCS Vol. 1214, 1997.

[Lia02] C. Liang. Compiler construction in higher order logic programming. In *4th International Symposium on Practical Aspects of Declarative Languages*, Springer-Verlag LNCS no. 2257, pages 47–63, 2002.

[Mil91a] D. Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269. MIT Press, June 1991.

[Mil91b] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[Mot00] M. Mottl. Automating functional program transformation. MSc Thesis. Division of Informatics, University of Edinburgh, September 2000.

[Muñ96] César Muñoz. Confluence and preservation of strong normalization in an explicit substitution calculus. In *Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 440–447. IEEE Computer Society Press, July 1996.

[Nad99] Gopalan Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), March 1999.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In Harald Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291. Springer-Verlag, July 1999.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NW98] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Tea02] The Coq Development Team. The Coq Proof Assistant Reference Manual Version 7.2. Technical Report 255, INRIA, February 2002. More recent versions may be obtained from the site `http://coq.inria.fr/`.

A Head Normalization with Eager Substitution

We provide concreteness here to the description in Section 4 of the head normalization procedure that makes only implicit use of the suspension notation. We employ Standard ML (SML) as our 'pseudo-code language' in this presentation. This choice is motivated by the fact that it permits the essential details of the procedure to be sketched while retaining succinctness in description. We assume a familiarity with SML to an extent that might be obtained from the tutorial introduction in [Har86], but the uninitiated reader will likely find the code we present self explanatory.

The first task is to provide datatype declarations for the terms and the closures that will be needed in the reduction procedure. This is done below:

```
datatype rawterm = const of string
 | var 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)
```

Notice 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 functions to, respectively, dereference a term and 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 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 *hn_eager* and *subst* whose definitions appear in Figures 3 and 4 implement each of these phases. The invocation of *hn_eager* 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 *hn_eager* will in general be a quadruple that is to be interpreted implicitly as a suspension. In reality, this suspension

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

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 *head_norm_eager* that is defined as follows:

```
fun head_norm_eager(t) = (hn_eager(t,0,0,nil,false) ; ())
```

That *head_norm_eager* is true to its intended purpose is the content of the following theorem:

Theorem 5 *Let t be a reference to the representation of a de Bruijn term that has a head normal form. Then head_norm_eager(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. Using an induction on the structure of terms, we can see that *head_norm_eager(t)* carries out a sequence of rewriting steps on t that corresponds first to a head reduction sequence as per Definition 3 of the suspension term encoded by t and then, possibly, a sequence of reading rule applications on the last term in the head reduction sequence. Proposition 4 guarantees that the head reduction sequence terminates. The reading relation is strongly normalizing. Thus *head_norm_eager(t)* must terminate. That t must be a reference when this happens to a head normal form for the term it originally represented now follows from the correctness of the rewrite rules in Figure 1.

□

B Head Normalization with Delayed Substitution

We are now interested in reflecting suspensions fully into term structure. The new datatype declarations appear below:

```
datatype rawterm = const of string
  | var 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)
```

Notice that these declarations permit terms of arbitrary form, as opposed to simply closures, to appear in environments.

The main work in this version of our head normalization routine is performed by the function *hn_lazy* defined in Figure 5. This function has a structure that is in many respects identical to that of the environment based procedure *hn_eager* seen in Appendix A; we note especially that the arguments of this procedure implicitly encodes a suspension and that it is also invoked in one of two modes that cause it to compute either a weak or a strong head normal form. There are, however, a few 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. A second difference arises from the fact that the new procedure must be prepared to also process suspensions. In order to preserve the ability to commit structures to heap only when necessary, the present procedure invokes the head normalization process on the embedded suspension. It is interesting to note that the sequence of rewriting steps that results is still encompassed by the head reduction sequences of Definition 3. Finally, in the case that a suspension has been processed in a mode intended to find a weak head normal form, it is possible for the returned value to itself be an (implicit) suspension; this would happen if the term reduces to one that is an abstraction at the top-level. This suspension must be made explicit and, further, it should be transformed into an abstraction using rule (r7) before computation can proceed. This effect is accomplished by an invocation of the *make_explicit* function defined below:

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

Figure 5: Head normalization with the capability of delaying substitutions

```
              ref(lam(ref(susp(t,ol+1,nl+1,dum(nl)::e))))
```

The function *hn_lazy* reduces terms that correspond implicitly to suspensions. The external interface to this function is provided by the following function:

```
  fun head_norm_lazy(t) = (hn_lazy(t,0,0,nil,false) ; ())
```

The correctness of *head_norm_lazy* is the content of the following theorem whose proof is similar to that of Theorem 5.

Theorem 6 *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 head_norm_eager(t) terminates and, when it does, t is a reference to the representation of a generalized head normal form of the original term.*

An observant reader may note that the functions *hn_eager* and *hn_lazy* that we have presented sometimes create new terms where this can be avoided—*e.g.*, when the term being processed is an application that is unchanged by the environment—and also miss out on some opportunities for sharing in reduction. These choices have been made here for clarity and brevity in presentation. In the actual implementations, care has been taken to ensure that all such extraneous aspects have a neutral impact on our experiments.