Tradeoffs in the Intensional Representation of Lambda Terms

Chuck Liang[1] and Gopalan Nadathur[2]

[1] Department of Computer Science, Hofstra University, Hempstead, NY 11550
Email: cscccl@hofstra.edu, Fax: 516-463-5790
[2] Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455
Email: gopalan@cs.umn.edu, Fax: 612-625-0572

Abstract. Higher-order representations of objects such as programs, specifications and proofs are important to many metaprogramming and symbolic computation tasks. Systems that support such representations often depend on the implementation of an intensional view of the terms of suitable typed lambda calculi. Refined lambda calculus notations have been proposed that can be used in realizing such implementations. There are, however, choices in the actual deployment of such notations whose practical consequences are not well understood. Towards addressing this lacuna, the impact of three specific ideas is examined: the de Bruijn representation of bound variables, the explicit encoding of substitutions in terms and the annotation of terms to indicate their independence on external abstractions. Qualitative assessments are complemented by experiments over actual computations. The empirical study is based on λProlog programs executed using suitable variants of a low level, abstract machine based implementation of this language.

1 Introduction

This paper concerns the representation of lambda terms in the implementation of programming languages and systems in which it is necessary to examine the structures of such terms during execution. The best known uses of this kind of lambda terms appears within higher-order metalanguages [16, 21], logical frameworks [9, 19] and proof development systems [5, 7, 20]. Within these systems and formalisms, the terms of a chosen lambda calculus are used as data structures, with abstraction in these terms being used to encode binding notions in objects such as formulas, programs and proofs, and the attendant β-reduction operation capturing important substitution computations. Although the intensional uses of lambda terms have often pertained to this kind of "higher-order" approach to abstract syntax, they are not restricted to only this domain. Recent research on the compilation of functional programming languages has, for example, advocated the preservation of types in internal representations [23]. Typed intermediate languages that utilize this idea [22] naturally call for structural operations on lambda terms during computations.

The traditional use of lambda terms as a means for computing permits each such term to be compiled into a form whose only discernible relationship to the original term is that they both reduce to the same value. Such a translation is, of course, not acceptable when lambda terms are used as data structures. Instead, a representation must be found that provides a rapid access at runtime to the *form* of a term and that also facilitates comparisons between terms based on this structure. More specifically, the relevant comparison operations usually 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. This control can be realized in principle by introducing a new category of terms that embody terms with 'suspended' substitutions. The detailed description of such an encoding is, however, a little complicated because the propagation of substitutions and the contraction of redexes inside the context of abstractions have, in general, to be considered when comparing terms.

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 [4]. 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 [1, 3, 10, 18]). 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.

This paper seeks to illuminate this empirical question. The vehicle for its investigation is the *Teyjus* implementation of λProlog [17]. λProlog is a logic programming language that employs lambda terms as a representational device and that, in addition to the usual operations on such terms, uses higher-order unification as a means for probing their structures. The *Teyjus* system, therefore, implements intensional manipulations over lambda terms. This system isolates several choices in term representation, permitting them to be varied and their impact to be quantified. We employ this concrete setting to understand three different issues: the value of explicit substitutions, the benefits and drawbacks of the de Bruijn notation and the relevance of an annotation scheme that determines the dependence of (sub)terms on external abstractions. Using a mixture of qualitative characterizations and experiments, we conclude that:

1. Explicit substitutions are useful so long as they provide the ability to combine β-contraction substitutions.

2. The potential disadvantage of the de Bruijn representation, namely the need to renumber indices during β-contraction, is not significant in practice.
3. Dependency annotations can improve performance, but their effect is less pronounced when combined with the ability to merge environments.

The rest of this paper is structured as follows. In the next section, we describe a notation for lambda terms that underlies the *Teyjus* implementation. This notation embodies an explicit representation of substitutions but one that can, with suitable control strategies, be used to realize substitutions either eagerly or lazily. 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 three sections discuss, in turn, the differences in lambda term representation of present interest and provide the results of our experiments. We conclude the paper with an indication of other questions that need to be examined empirically.

2 A Notation for Lambda Terms

We use an explicit substitution notation for lambda terms in this paper that builds on the de Bruijn method for eliminating bound variable names. A notation of this kind conceptually encompasses two categories of expressions, one corresponding to terms and the other corresponding to environments that encode substitutions to be performed over terms. In a notation such as the $\lambda\sigma$-calculus [1] that use exactly these two categories, an operation must be performed on an environment expression each time it is percolated inside an abstraction towards modifying the de Bruijn indices in the terms whose substitution it represents. The notation that we have designed for use in the implementation of λProlog instead allows these adjustments to be carried out in one swoop when a substitution is actually effected rather than in an iterated manner. To support this possibility, this notation includes a third category of expressions called environment terms that encode terms together with the 'abstraction context' they come from. Our 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 our *annotated suspension notation* are given by the following rules:

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

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 the category of natural numbers. Terms correspond to lambda terms. In keeping with the de Bruijn scheme, $\#i$ represents a variable bound by the ith abstraction looking back from the occurrence. An expression of the form $[\![t, ol, nl, e]\!]_o$ 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 e and its remaining bound variables being renumbered to reflect the fact that t used to appear within ol abstractions but now appears within nl of them. Conceptually, the elements of an environment are either substitution terms generated by a contraction or are dummy substitutions corresponding to abstractions that persist in an outer context. However, renumbering of indices may have to be done during substitution, and to encode this the environment elements are annotated by a relevant abstraction level. To be deemed well-formed, suspensions must satisfy certain constraints that have a natural basis in our informal understanding of their content: in an expression of the form $[\![t, i, j, e]\!]_o$ 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.

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.

The rule (r2) pertaining to 'reading' an instantiatable variable is based on a particular interpretation of such variables: substitutions that are made for them must not contain de Bruijn indices that are captured by external abstractions. This is a common understanding of such variables but not the only one. For example, treating these variables as essentially first-order ones whose instantiation can contain free de Bruijn indices provides the basis for lifting higher-order unification to an explicit substitution notation [8]. We comment briefly on this possibility at the end of the paper but do not treat it in any detail here.

The correctness of some of our reading rules, in particular, the rules (r8)-(r10), depends on consistency in the use of annotations. Our assumption is that these are correctly applied at the outset; thus, the static (compilation) process that creates initial internal representations of terms is assumed to apply the annotation c to only those terms that do not have unbound de Bruijn indices in them. It can then be seen that the rewrite rules preserve consistency while also attempting to retain the content in annotations [15].

The ultimate utility of our notation is dependent on its ability to simulate reduction in the lambda calculus. That it is capable of doing this can be seen

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

Fig. 1. Rule schemata for rewriting annotated terms

in two steps [15]. First, underlying every consistently annotated term in the suspension notation is intended to be a de Bruijn term that is to be obtained by 'calculating out' the suspended substitutions. The reading rules can be shown to possess properties that support this interpretation: every sequence of rewritings using these rules terminates and any two such sequences that start at the same term ultimately produce annotated forms of the same de Bruijn term. It can then be shown that the de Bruijn term t β-reduces to s if and only if any t' that is a consistently annotated version of t can be rewritten using our rules to a term s' that is itself a consistently annotated version of s.

The (β_s') rule is redundant to our collection if our sole purpose is to simulate β-contraction; indeed, omitting this rule yields a calculus that is similar to those in [3] and [10]. However, the (β_s') rule is the only one in our system that permits substitutions arising from different contractions to be combined into one environment and, thereby, to be carried out in the same walk over the structure of the term being substituted into. The rule (r11) is also redundant, but it serves a similar useful purpose in that it that it allows a reduction to be combined with a renumbering walk after a term has been substituted into a new (abstraction) context. In fact, the main uses of rules (r11) and (r12) arise right after a use of rule (r5) and a reduction procedure based on our rules can actually roll these distinct rule applications into one. Rather than eliminating the effects of rules (β_s') and (r11), it is possible to replace them with more general ones that are capable of merging the environments in *any* term of the form $[\![[\![t, ol_1, nl_1, e_1]\!]_u, ol_2, nl_2, e_2]\!]_v$ to produce an equivalent term of the form $[\![t, ol, nl, e]\!]_v$; such a collection of rules is, in fact, presented in [18]. However, embedding this larger collection of rules in an actual procedure can be difficult and an underlying assumption here is that the rules (β'_s) and (r11) provide an efficient way to capture most of their useful effects. A final observation is that the rules (r8)-(r10) are redundant in a sense analogous to the (β'_s) rule. However, using these rules can have practical benefits as we discuss in Section 6.

3 Computations in λProlog

The language λProlog is one that, from the perspective of this paper, 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. These lambda terms are complemented by a notion of equality given by the α-, β- and η-conversion rules. Second, λProlog uses a unification operation that builds in the extended notion of equality accompanying lambda terms. Finally, the language incorporates two new kinds of goals that provide, respectively, for scoping over names and clauses defining predicates.

The new features of λProlog endow it with useful metalanguage capabilities based essentially on the fact that the abstraction construct that is present in lambda terms provides a versatile mechanism for capturing the binding notions that appear in a variety of syntactic constructs. Suppose, for instance, that we are interested in representing a formula of the form $\forall x\ P(x)$ where $P(x)$ represents a possibly complex formula in which x appears free. Using lambda terms, this formula may be encoded as $(all\ (\lambda x\, \overline{P(x)}))$, where all is a constant chosen to encode the predicative force of the universal quantifier and $\overline{P(x)}$ represents $P(x)$. This kind of explicit treatment of quantifier scope has several advantages. Identity of formulas under variable renaming is implicit in the encoding; the representations of $\forall x\ P(x)$ and $\forall y\ P(y)$ are, for example, equivalent lambda terms. Quantifier instantiation can be realized uniformly through application. Thus, the instantiation of the quantifier in $(all\ Q)$, where Q itself is a complex structure possibly containing quantifiers, by the term t can be realized simply by writing down the expression $(Q\ t)$; β-contraction realizes the actual substitution with all the renamings necessary for logical correctness. Expressions with instantiatable variables can be used in conjunction with the enhanced unification notion to analyze formulas in logically sophisticated ways. Finally, many computations on formulas are based on recursion over their structure. The usual Horn clauses of a logic programming language already provide for a recursion over first-order structure. The new scoping mechanisms of λProlog complement this to realize also a recursion over *binding* structure in an elegant and logically justifiable way.

The capabilities of λProlog that we have outlined above have been exploited for a variety of purposes in the past, ranging from building theorem provers and encoding proofs to implementing experimental programming languages. A detailed discussion of these applications is beyond the scope of this paper. For our present purposes it suffices to note that *all* of them depend on the ability to perform conversion operations on lambda terms, to compare their structures and to decompose them in logical ways. Thus, λProlog provides a rich programming framework—that is not unlike others that utilize lambda terms intensionally—for studying the impact on performance of choices in the representation of these terms. In the following sections, we use this framework in quantifying some of these differences. We identify below a taxonomy of λProlog computations that is based roughly on the 'level' of the higher-order feature used and we use this to describe a collection of testing programs to be employed in our empirical study. The test suite is available from the Teyjus site at `http://teyjus.cs.umn.edu/`.

First-Order Programs. This category exercises no higher-order features and should therefore be impervious to differences in the way (lambda) terms are treated. We include two programs from this category:

- *[quicksort]* A standard Prolog implementation of the familiar sorting routine.
- *[pubkey]* An implementation of a public key security protocol described in [6]. This program uses the new scoping devices in λProlog.

First-Order Like Unification with Reduction. Genuine lambda terms may appear in programs in this category, but these figure mainly in reduction computations, most unification problems being first-order in nature. Matching the representation of a lambda term with the pattern *(all Q)*, for instance, yields a unification problem of this kind and the subsequent application *(Q t)* generates a reduction. The two programs included in this category are the following:

- *[hnorm]* A head normalization routine used to reduce a collection of randomly generated lambda terms.
- *[church]* A program that involves arithmetic computations with Church numerals and associated combinators.

L_λ Style Programs. 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 controlled form of reduction—the argument of a redex is always a constant—and a restricted form of higher-order unification [12]. We test the following programs in this category:

- *[typeinf]* A program that infers type schemes for ML-like programs.
- *[compiler]* A compiler for a small imperative language [11].

Unrestricted Programs. Programs in this class make essential use of (general) higher-order unification. As such, they tend to involve significant backtracking and they also encompass β-reduction where the arguments of redexes are complex terms. The programs tested in this category are the following:

- *[hilbert]* A λProlog encoding of Hilbert's Tenth Problem [13].

– *[funtrans]* A transformer of functional programs [14].

The *Teyjus* system used in our experiments is an abstract machine and compiler based implementation of λProlog. It uses a low-level encoding of the terms in the annotated suspension notation. The use of the rewrite rules in Figure 1 within *Teyjus* are confined to a procedure that transforms terms to head normal forms, upon which all comparison operations are based. 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.

4 The Value of Explicit Substitutions

Reflecting an explicit substitution notation into the representation of lambda terms provides the basis for a lazy strategy in effecting reduction substitutions. There are two potential benefits to such laziness.

First, actual substitutions may be delayed till a point where it becomes evident that it is unnecessary to perform them. Thus, consider the task of determining if the two terms $((\lambda\,\lambda\,\lambda\,(\#3\ \#2\ s))\ (\lambda\,\#1))$ and $((\lambda\,\lambda\,\lambda\,(\#3\ \#1\ t))\ (\lambda\ \#1))$ are identical, assuming that s and t are some unspecified but complicated terms. We can conclude that they are not by observing that they reduce respectively to $\lambda\,\lambda\,(\#2\ s')$ and $\lambda\,\lambda\,(\#1\ t')$, where s' and t' are terms that result from s and t by appropriate substitutions. Note that, in making this determination, it is not necessary to explicitly calculate the results of the substitutions over the terms s and t. The annotated suspension notation and a suitable head-normalization procedure [15] provide the basis for such an approach.

Second, such laziness makes it possible to combine substitution walks that arise from contracting different β-redexes. Thus, suppose that we wish to instantiate the two quantifiers in the formula represented by $(all\ (\lambda x\,(all\ (\lambda y\,P))))$, where P represents an unspecified formula, with the terms t_1 and t_2. Such an instantiation is realized through two reductions, eventually requiring t_2 and t_1 to be substituted for the first and second free variables in P and the indices of all other free variables to be decremented by two. All these substitutions involves a walk over the *same* structure—the structure of P—and it would be profitable if they could all be done together. To combine walks in this manner, it is necessary to temporarily suspend substitutions generated by β-contractions. In a situation in which all the redexes are available in a single term, this kind of delaying of substitution can be built into the reduction procedure through *ad hoc* devices. However, in the case being considered, the two quantifier instantiations are ones that can only be considered incrementally and, further, intervening structure needs to be processed before the abstraction giving rise to the second redex is encountered. The structure that engenders sharing is therefore not all available within a single call to a reduction procedure and an explicit encoding of substitution over P seems to be necessary for realizing this benefit.

Towards understanding how important these factors are in practice, different strategies were experimented with in the head normalization routine in *Teyjus*. Three variations were tried. In one case, each time a β_s rule was used, the effects

Program	*Eager*		*Lazy without Merging*		*Lazy with Merging*	
	Running Time	*Reading Rule Applications*	*Running Time*	*Reading Rule Applications*	*Running Time*	*Reading Rule Applications*
quicksort	0.25 secs	0	0.25 secs	0	0.25 secs	0
pubkey	0.34 secs	0	0.34 secs	0	0.33 secs	0
church	0.36 secs	243461	0.39 secs	367314	0.27 secs	73200
hnorm	0.75 secs	266970	0.83 secs	439121	0.66 secs	89020
typeinf	14.70 secs	10291085	14.81 secs	17582708	9.58 secs	2177884
compiler	3.53 secs	2496431	3.82 secs	3391088	2.26 secs	318703
hilbert	0.48 secs	296027	0.36 secs	58894	0.34 secs	5489
funtrans	2.20 secs	44803	2.22 secs	60116	2.19 secs	24290

Fig. 2. Comparison of Reduction Substitution Strategies

of the resulting substitution were immediately calculated out, mimicking the customary eager approach. The second version used lazy substitution but did not use the (β_s') rule in the rewriting process. This version gives us the benefits of delayed substitution *without* the added advantage of combining substitution walks. The last variation used the full repertoire of rewrite rules, once again in a demand-driven mode for calculating substitutions, thereby attempting to draw on all the benefits of explicit substitutions. The overall framework in all three cases was provided by a graph-based reduction routine with an iterative control and that uses a stack as an auxiliary store. Such a procedure writes the result of each rewriting step based on the rules in Figure 1 to the heap. An alternative procedure, that was not utilized but that is applicable equally to all three variants tested and hence likely to produce similar timing behaviour, would embed most of the information written to heap within a recursion stack.

Figure 2 tabulates the running time and the number of applications of the rewrite rules that percolate the effects of substitutions over terms for each variant over the suite of programs described in Section 3. All the tests were conducted on a 400MHz UltraSparc and each figure in the table represents the average of five runs. The data in the table indicates a clear preference for a lazy approach to substitution combined with merging of substitutions in the cases where lambda terms are used intrinsically in computation. In the cases where the computation predominantly involves reduction or L_λ style processing, the time improvements range from 12% to over 35%. The measured time is over *all* computations, including backchaining over logic programming clauses. The difference attributable to the substitution strategy is pinpointed more dramatically and accurately by the counts of reading rule applications that are directly responsible for the timing differences and that are unaffected by the specifics of term representation and reduction procedure implementation. These figures indicate a reduction of structure traversal to between a third and an eigth in the cases mentioned through the use of lazy substitution with merging. A complementary observation is that while the number of β-redexes contracted remains unchanged, between 50% and 90% of these contractions are realized via the (β_s') rule.

We note that behavior degrades in some cases when lazy substitution is used without merging. There are two explanations for this. First, in this mode, terms of the form $[\![t, ol, nl, e]\!]_v$ where t is itself a suspension are often encountered. In a demand driven approach the two suspensions have to be worked inwards in tandem and this has a noticeable overhead. Second, backtracking, a possibility in a logic programming like setting, has an impact. Backtracking causes a revocation of the changes that have been made beyond the choice point that computation is retracted to. By carrying out substitution work that is deterministic late, a source of redundancy is introduced. This effect is evident in the increase in reading rule applications under a lazy substitution strategy without merging.

The data tabulated above pertain to the situation when annotations are used. Without these the differences are more dramatic as we discuss later. The conclusion from our observations thus seems to be that explicit substitutions are important for performance provided they are accompanied by merging.

5 The Treatment of Bound Variables

The common approaches to representing bound variables can be separated into two categories: those that use explicit names or, roughly equivalently, pointers to cells corresponding to binding occurrences and those that use de Bruijn indices. In comparing these approaches, it is important to distinguish two operations to which bound variables are relevant. 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*. Under the de Bruijn scheme, this context is implicitly present in the numbering of bound variables, obviating the explicit attachment of the abstractions.

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, *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.*, $\hat{t}_2$.[1] 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 [2], is to always perform a renaming and, more significantly, to fold this into the same structure traversal as that realizing 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 where an immediate simplification by one of the rules (r8)-(r10) or (r12) is not possible.

Figure 3 tabulates the data gathered towards this end for the programs in our test suite that use reductions. An interesting observation is that *no* renumbering is actually involved in the case of L_λ style programming. The reason for this is not hard to see—the only reductions performed are those corresponding to 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 but adopting a merging based

[1] 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	*Eager*		*Lazy with Merging*	
	Total Substitutions	*Renumbering Substitutions*	*Total Substitutions*	*Renumbering Substitutions*
church	34999	3267	34723	12
hnorm	25497	0	32003	0
typeinf	777832	0	722291	0
compiler	154967	0	100519	0
hilbert	3840	1500	1539	411
funtrans	8587	146	7374	19

Fig. 3. Renumbering with the de Bruijn Representation

approach to substitution can reduce this considerably. 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 necessarily constitute an extra cost. In general, when a term is substituted in, it is necessary also to examine its structure and possibly 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 [2] 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. An indication of the loss in sharing can be obtained from the differences in the number of (β_s) and (β'_s) reductions under the two strategies in those cases where renumbering is an issue. Our measurements show that, under an 'umbrella' regime of delayed substitution with merging, these numbers were *identical* in all the relevant cases. Thus, there is no loss in sharing from combining the reduction and renumbering walks and, consequently, no real renumbering overhead relative to our test suite.

Our conclusion, then, is 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 and, in fact, representations of the latter kind are not serious contenders from this perspective in low-level implementations. The de Bruijn scheme has the drawback of a renumbering overhead in realizing β-contraction. However, our experiments show that this overhead is either negligible or nonexistent in a large number of cases.

6 The Relevance of Annotations

Annotations that are included with terms have the potential for being useful in two different ways. First, they can lead to substitutions walks being avoided when it is known that the substitutions will not affect the term. Second, by allowing a suspension to be simplified directly to its skeleton term, they can lead to a preservation of sharing of structure, and, hence, of reduction work, in a graph based implementation. Both effects are present in the rules (r8)-(r10) in Figure 1, the only ones to actually use annotations. There is, of course, a cost associated with maintaining annotations as manifest in the structure of all the other rules. 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 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 that ignores the structure of the term suffices in utilizing the information in the annotation.

Program	*Eager*		*Lazy with Merging*	
	Without Annotations	*With Annotations*	*Without Annotations*	*With Annotations*
quicksort	0.26	0.25	0.25	0.25
pubkey	0.33	0.34	0.31	0.33
church	1.90	0.36	0.29	0.27
hnorm	0.77	0.75	0.68	0.66
typeinf	48.56	14.70	9.56	9.58
compiler	19.51	3.53	2.29	2.26
hilbert	0.60	0.48	0.34	0.34
funtrans	2.20	2.20	2.26	2.19

Fig. 4. The Effect of Annotations

Experiments were conducted to quantify the benefits of annotations. The different versions of the head normalization routine described in Section 4 were modified to ignore annotations. The chosen programs were then executed on a 400MHz UltraSparc using the various versions of the *Teyjus* system. Figure 4 tabulates the running time in seconds for four of these versions; we have omitted the case of delayed substitutions without merging since the results are similar to the version with eager substitutions. The indication from these data is that annotations can make a significant difference in a situation where substitution is performed eagerly—for example, the running time is reduced by about 70% and 80% in the case of the two L_λ style programs—but they have little effect in the situation when lazy substitution with merging is used. Interpreting these observations, it appears that annotations can lead to the recognition of a large amount of unnecessary substitution work. In the situation where substitution walks can be merged, however, these can also be combined with walks for reducing the term. Since reduction has to be performed in any case, the redundancy when annotations are not used is minimal. Consistent with the observations from the previous section, the data for the case of lazy substitutions also indicates little benefit from shared reduction.

7 Conclusion

We have examined the tradeoffs surrounding the use of three ideas in the machine encoding of lambda terms. Our study indicates that a notation that supports a delayed application of substitution as well as a merging of substitution walks can be used to significant practical effect. Within this context the de Bruijn scheme for treating bound variables has definite advantages with little attendant costs. Finally, the benefits of annotations appear to be marginal at best when reduction and substitution are performed in tandem. Our observations have been made relative to a graph-based approach to representing terms. It is further possible to use hash-consing in the low-level encoding as has been done in the FLINT system that also employs the annotated suspension notation [22]. While we have not experimented with this technique explicitly, we believe that its use will be neutral to the other choices considered here.

The work reported here can be extended in several ways. One further question 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. Along a different direction, the implementation of higher-order unification needs to be considered more carefully. By changing the interpretation of instantiatable or meta variables, it is possible to lift higher-order unification to an explicit substitution notation. Doing so has the benefit of making the application of substitutions to meta variables very efficient. However, there are also costs: a more general mechanism for combining substitutions is needed and context information must be retained dynamically to translate metavariables prior to the presentation of output. There is a tradeoff here that can, once again, be assessed empirically. A final question concerns a more refined set of benchmarks, one that makes finer distinctions in the categories of computations on lambda terms. Such a refinement may reveal further factors that can influence the tradeoffs in representations.

8 Acknowledgements

This work was supported in part by NSF Grant CCR-0096322. Useful presentation suggestions were received from the referees, not all of which could be explicitly utilized due to space limitations.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. L. Aiello and G. Prini. An efficient interpreter for the lambda-calculus. *The Journal of Computer and System Sciences*, 23:383–425, 1981.
3. 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.

4. 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.
5. R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
6. G. Delzanno. Specifying and debugging security protocols via hereditary Harrop formulas and λProlog - a case-study. In *Fifth International Symposium on Functional and Logic Programming.* Springer Verlag LNCS vol. 2024, 2001.
7. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993.
8. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.
9. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
10. F. Kamareddine and A. Ríos. Extending the λ-calculus with explicit substitution which preserves strong normalization into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, 1997.
11. C. Liang. Compiler construction in higher order logic programming. In *4th International Symposium on Practical Aspects of Declarative Languages*, pages 47–63. Springer Verlag LNCS No. 2257, 2002.
12. D. Miller. A logic programming language with λ-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
13. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.
14. M. Mottl. Automating functional program transformation. MSc Thesis. Division of Informatics, University of Edinburgh, September 2000.
15. G. Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), March 1999.
16. G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.
17. G. Nadathur and D. J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In *Automated Deduction–CADE-16*, pages 287–291. Springer-Verlag LNAI no. 1632, 1999.
18. G. Nadathur and D. S. Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.
19. C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer-Verlag LNCS 664, 1993.
20. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer Verlag, 1994.
21. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206. Springer-Verlag LNAI 1632, 1999.
22. Z. Shao. Implementing typed intermediate language. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323. ACM Press, September 1998.
23. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192. ACM Press, 1996.