An Explicit Substitution Notation in a λProlog Implementation

Gopalan Nadathur

Department of Computer Science
University of Chicago
Ryerson Hall, 1100 E 58th Street
Chicago, IL 60637

Phone: (773)-702-3497
Fax: (773)-702-8487
Email: `gopalan@cs.uchicago.edu`

1 Introduction

This abstract has a pragmatic intent: it explains the use of an explicit substitution notation in an implementation of the higher-order logic programming language λProlog. The particular aspects of this language that are of interest here are its provision of typed lambda terms as a means for representing objects and of higher-order unification as a tool for probing the structures of these objects. There are many uses for these facilities originating from the fact that they lead to direct and declarative support for a *higher-order abstract syntax* view of objects such as formulas and programs [MN87, PE88]. Detailed discussions of applications can be found in the literature, e.g. see [Fel93, HM92, NM94, Per91, Pfe88]. Success encountered in these various experiments has driven an effort on our part towards developing a good implementation of the language. An important ingredient of such an implementation is, of course, a sensible treatment of lambda terms.

The use that is made of lambda terms in λProlog is similar to that in other recent metalanguages and logical frameworks such as Isabelle [Pau94] and Elf [Pfe91] but differs significantly from they way these terms are used in functional programming languages. A suitable internal representation for these terms must, correspondingly, satisfy certain constraints that do not arise within functional programming. To begin with, computations typically need to compare terms, and their structures must, therefore, be accessible at runtime. The relevant comparison operations usually ignore bound variable names and, for this reason, equality modulo α-conversion must 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. At a level of detail, it is best to interleave the performance of reduction substitutions with comparison operations: both operations involve the same walks over term structures and the first indication of differences in structure makes it redundant to calculate the full effects of substitution. However, it is *essential* to be able to propagate reduction substitutions *under* abstractions in order to carry out the needed comparisons.

In previous work, we have developed an explicit substitution notation for the lambda calculus that provides a basis for meeting several of these requirements [NW90, NW97]. Our notation, called the *suspension notation*, is similar in spirit to the $\lambda\sigma$-calculus [ACCL91] but represents a concurrent and independent development. We have subsequently simplified and refined our notation to yield a version of it called the *annotated suspension notation* that is amenable to a direct use in implementations [Nad96]. A salient feature of the latter notation is the presence in it of annotations that indicate whether or not terms can be affected by substitutions generated by contracting external β-redexes. These annotations are useful in that they permit substitutions to be carried out trivially in certain situations. Effecting substitutions in this manner can also have other benefits: it can lead to a conservation of space and can foster a greater sharing of work in a graph-based implementation of reduction. Benefits such as these have been noted to be significant in practice [BR91].

In the rest of this abstract, we describe the annotated suspension notation, present a stack based procedure for head-normalizing terms in this notation and explain how these aspects fit into an implementation of higher-order unification. All the devices sketched here have been used in a C-based implementation of an abstract machine for λProlog.

2 The Annotated Suspension Notation

An explicit substitution notation involves at least two syntactic categories: those corresponding to terms and environments. Our notation, unlike the $\lambda\sigma$-calculus, does not require elements of the environment to be modified each time the environment is propagated into a new context. Rather, this task is carried out in one swoop when an actual substitution is effected. However, to support this possibility, environment elements must encode the 'context' they come from. This introduces a third syntactic category, that of *environment terms*. Formally, our terms, environments and environment terms are given by the following syntax rules:

$$
\begin{array}{lll}
\langle ATerm\rangle & ::= & \langle Cons\rangle \mid \langle Var\rangle \mid \#\langle Index\rangle \mid (\langle ATerm\rangle\ \langle ATerm\rangle)_o \mid \\
& & (\langle ATerm\rangle\ \langle ATerm\rangle)_c \mid (\lambda_o\,\langle ATerm\rangle) \mid (\lambda_c\,\langle ATerm\rangle) \mid \\
& & [\![\langle ATerm\rangle, \langle Nat\rangle, \langle Nat\rangle, \langle AEnv\rangle]\!]_o \mid [\![\langle ATerm\rangle, \langle Nat\rangle, \langle Nat\rangle, \langle AEnv\rangle]\!]_c \\
\langle AEnv\rangle & ::= & nil \mid \langle AETerm\rangle :: \langle AEnv\rangle \\
\langle AETerm\rangle & ::= & @\langle Nat\rangle \mid (\langle ATerm\rangle, \langle Nat\rangle)
\end{array}
$$

In these rules, $\langle Cons\rangle$ and $\langle Var\rangle$ stand for predetermined sets of constant and free variable symbols, $\langle Index\rangle$ is the category of positive numbers and $\langle Nat\rangle$ is the category of natural numbers. Our notation is based on the de Bruijn representation of lambda terms and, in this context, $\#i$ corresponds to a variable bound by the ith abstraction looking back from the occurrence. Our choice of the de Bruijn notation is motivated by the need to consider α-convertibility. An expression of the form $[\![t, ol, nl, e]\!]_o$ or $[\![t, ol, nl, e]\!]_c$, referred to as a *suspension*, constitutes a term with a 'suspended' substitution. Intuitively, this corresponds to the term t whose first ol variables have to be substituted for in the way determined by e and whose remaining bound variables have to be renumbered to reflect the fact that t used to appear within ol abstractions but now appears within nl of them; ol and nl are referred to as the old and new embedding levels and e constitutes the environment. Finally, all the 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 this information is not available.

Our notation includes, as usual, a collection of rewrite rules whose ultimate purpose is to simulate β-reduction. These rules or, rather, rule schemata, are presented in Figure 1. The interpretation of most schema variables in these rules should be obvious. The symbols v and u that are used for annotations are schema variables that can be substituted for by either c or o. Of these rules, the ones labelled (β_s) and (β_s') generate the substitution corresponding to the β-contraction rule on de Bruijn terms and the rules (r1)-(r13), referred to as the *reading rules*, serve to actually carry out such a substitution. The rules (β_s') and (r9)-(r13) are redundant if our sole purpose is to simulate β-contraction. However, using these rules can have practical benefits. Without (β_s'), it is impossible to combine substitution walks for different β-contractions. The rules (r9)-(r13), first of all, permit substitutions to be carried out trivially in certain cases. Moreover, these rules facilitate a sharing of reduction in a graph-based implementation. For example, using the rule (r13) wherever possible permits the sharing present in lazy reducers [HM76] to be matched in the production of weak head normal forms.

The syntax for expressions as presented above is not sufficiently restricted to correspond to situations of actual interest. For example, a term may be annotated with c even when it contains a variable occurrence bound by an external abstraction. Similarly, in a term of the form $[\![t, ol, nl, e]\!]_u$, the 'length' of e may be distinct from ol. Notions of consistency of annotation and wellformedness are presented in [Nad96] to preclude such situations. For our present purposes, it suffices to only note that if we begin a rewriting process with an (annotated) de Bruijn term, then every intermediate subexpression satisfies these constraints and that we can therefore assume that we are dealing with only such expressions. Now, underlying every (satisfactory) term in the annotated suspension notation is intended to be a de Bruijn term that is to be obtained by 'calculating out' the suspended substitutions using the reading rules. The reading rules possess properties that support this interpretation: Every sequence of rewritings using these rules terminates. Further, any two such sequences that start at the same term ultimately produce annotated forms of de Bruijn terms that differ at most in their annotations. These properties are established in [Nad96] and they provide the basis for the following definition:

Definition 1 *If t is a term in the annotated suspension notation, then $|t|$ denotes the de Bruijn term that is obtained by dropping the annotations from the normal form of t relative to the reading rules.*

A final observation concerns the status of free variables in terms. While substitutions are permitted for such variables, these much satisfy the usual non-capture restrictions in the lambda calculus. In particular, terms replacing them *must not* contain variables that could be bound by external abstractions. This restriction is manifest in rule (r2). This interpretation differs from the one in [DHK95] for such 'meta variables'. The latter interpretation can also be enforced here by dropping rule (r2) and by including more general

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

(β'_s) $\quad ((\lambda_u\, [\![t_1, ol+1, nl+1, @nl :: e]\!]_o)\ t_2)_v \rightarrow [\![t_1, ol+1, nl, (t_2, nl) :: e]\!]_v$

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

(r2) $\quad [\![x, ol, nl, e]\!]_u \rightarrow x,$
provided x is a free variable.

(r3) $\quad [\![\#i, 0, nl, nil]\!]_u \rightarrow \#(i + nl).$

(r4) $\quad [\![\#1, ol, nl, @l :: e]\!]_u \rightarrow \#(nl - l).$

(r5) $\quad [\![\#1, ol, nl, (t, l) :: e]\!]_u \rightarrow [\![t, 0, nl - l, nil]\!]_u.$

(r6) $\quad [\![\#i, ol, nl, et :: e]\!]_u \rightarrow [\![\#(i-1), ol-1, nl, e]\!]_u,$
provided $i > 1$.

(r7) $\quad [\![(t_1\ t_2)_u, ol, nl, e]\!]_v \rightarrow ([\![t_1, ol, nl, e]\!]_v\ [\![t_2, ol, nl, e]\!]_v)_v.$

(r8) $\quad [\![(\lambda_u\, t), ol, nl, e]\!]_v \rightarrow (\lambda_v\, [\![t, ol+1, nl+1, @nl :: e]\!]_o).$

(r9) $\quad [\![(t_1\ t_2)_c, ol, nl, e]\!]_u \rightarrow (t_1\ t_2)_c.$

(r10) $\quad [\![(\lambda_c\, t), ol, nl, e]\!]_u \rightarrow (\lambda_c\, t).$

(r11) $\quad [\![[\![t, ol, nl, e]\!]_c, ol', nl', e']\!]_u \rightarrow [\![t, ol, nl, e]\!]_c.$

(r12) $\quad [\![[\![t, ol, nl, e]\!]_o, 0, nl', nil]\!]_o \rightarrow [\![t, ol, nl + nl', e]\!]_o.$

(r13) $\quad [\![t, 0, 0, nil]\!]_u \rightarrow t$

Figure 1: Rule schemata for rewriting annotated suspensions

rules for merging environments [NW97]. However, our particular approach to implementing higher-order unification is based on the more conservative interpretation of variables.

3 Reduction to Head Normal Form

The comparison of lambda terms as required in higher-order unification is usually based on their head normal forms. This notion can be lifted to the annotated suspension notation.

Definition 2 *An annotated suspension term is in head normal form if it has the structure*

$$(\lambda_{u_1} \ldots (\lambda_{u_n} (\ldots(h\ t_1)_{u_{n+1}}\ \ldots\ t_m)_{u_{n+m}}) \ldots)$$

where, for $1 \leq i \leq (n+m)$, u_i *is either* o *or* c *and* h *is either a constant or a variable reference. In this case,* $t_1, \ldots, t_m$ *are called its arguments,* h *is called its head and* n *is its binder length.*

Our notion of head normal forms is useful because it is transparently related to head normal forms in the de Bruijn notation.

Theorem 3 *Let* t *be an de Bruijn term. Further suppose that the rules in Figure 1 allow it to be rewritten to a head normal form in the sense of Definition 2 that has* h *as its head,* n *as its binder length and* $t_1, \ldots t_m$ *as its arguments. Then* t *has the term*

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

as a head normal form in the conventional sense.

Proof. See [Nad96].

□

Thus, a procedure for reducing annotated suspension terms to head normal forms is of interest in implementing comparison operations. The rewrite rules in Figure 1 can be reflected into a procedure for carrying out such a reduction. Appendix A contains type declarations in C for representing annotated suspension expressions and also presents a stack based approach for affecting the desired transformation. The two main procedures in this code are *head_norm* and *lazy_read* that serve, respectively, to reduce a term to head normal form and to expose a non-suspension structure for a given suspension term. These procedures use a reduction stack and an auxiliary stack for remembering applications. In the intended use, *head_norm* is invoked with the term to be reduced appearing as the top element of the reduction stack. On termination of *head_norm*, the variable *numabs* is set to the binder length of the head normal form that is produced, and the reduction stack contains the head and the arguments in successive locations below the top of stack. (A separate application stack is used only so that the head normal form can to be presented in this manner.) Some interesting aspects of our procedures are that they are graph-based and utilize destructive changes in implementing reduction, and sharing results in them only through the use of the rewrite rules (r9)-(r13). Space considerations preclude a detailed comparison here with other reduction procedures.

Theorem 4 *Let t be (the representation of) a de Bruijn term and let head_norm be invoked with a pointer to t placed on the top of the reduction stack. Then head_norm terminates whenever t has a head normal form. Further, if it terminates with numabs set to n and with pointers to $h, t_1, \ldots, t_m$ appearing in successive locations below the top of the stack, then t has a head normal form with binder length n, head h and arguments $t_1, \ldots, t_m$.*

Proof. Only a sketch is provided. First, it can be seen that a head normal form exists in our sense if and only if one exists in the usual sense. Second, the notion of a head reduction sequence can be generalized to our notation and, using correspondences between the rules in Figure 1 and β-contraction in the lambda calculus, it can be shown that such a sequence terminates if and only if a head normal form exists. This generalized notion actually incorporates sharing in reduction and is defined and studied in detail in [Nad96]. Finally a correspondence can be established between the iteration in *head_norm* and *lazy_read* and the terms in a head reduction sequence. Termination and the production of an actual head normal form follow from these observations.

□

The presentation of programs in Appendix A strikes a balance between efficiency and perspicuity. Our actual implementation incorporates several improvements. First, our term representation is really a low level rendition of the structure and union presentations. In this representation, the tag and annotation fields are reflected into bit positions and efficient (combined) manipulations of these fields can be realized through the use of bit masks. Second, we have used representation based on the Böhm tree structure, choosing to combine several applications into one structure. Such a representation can conserve space and also has advantages from the perspective of compiling comparison and unification operations. Finally, new term and environment structures are created in an internally managed heap not through the repeated invocation of *malloc*.

4 Suspension Notation in Higher-Order Unification

Our implementation of higher-order unification is based in essence on Huet's procedure [Hue75]. A unification problem is represented within this procedure by a set of pairs of typed lambda terms: the objective is to find substitutions for free variables that result in the two terms in each pair becoming identical. An important step in the procedure is that of reducing the terms in each pair to head normal form. Three different kinds of actions are possible depending on the structure of these forms:

1. If both terms have equal length binders and their heads are identical constants or identical de Bruijn indices, then the pair is replaced by pairs of corresponding arguments.

If both heads are constants or de Bruijn indices, but are distinct, then a failure in this branch of unification is registered. (Distinctions in binder length and the η-conversion rule are discussed below.)

2. If one of the heads is a free variable and the other is a constant or a de Bruijn index, then one of several (closed) substitutions must be considered for the free variable. The structure of these substitutions is determined completely by the types of the free variable that is the head of one term and the constant that is possibly the head of the other. Note that there are choices in this action that could lead to branching in unification.

3. If the heads of both terms are free variables, then a consideration of this pair is delayed.

There is actually a preferred order in the application of these actions: all pairs satisfying the first description are treated first—and this includes even the pairs added as a result of this action—before pairs are treated under the second classification.

The procedure outlined above can be lifted to our notation by exploiting the correspondence of head normal forms under it with that under the de Bruijn notation. The practical consequence of this is that the performance of reduction substitution can be interleaved with the unification procedure. The reduction procedure discussed in the previous section provides a convenient basis for realizing this idea. In particular, the procedure can be invoked for each term in a pair, taking care to preserve the structure it produces in the stack in each case. A simple way to realize this is to not reset the top of reduction stack after the first term is reduced. The values left in the stack can then be used directly in realizing the needed actions.

The notion of equality for lambda terms in λProlog and in most other situations of interest includes convertibility by virtue of the η-rule. The annotated suspension notation provides an elegant mechanism for handling this rule on the fly. Observe, first, that the embedding of a term t under n abstractions can be represented simply by the term $[\![t, 0, n, nil]\!]_o$. Now, in the case that t is a constant or a de Bruijn index, the result of this substitution is easy to calculate: it is simply the constant itself or the index 'bumped up' by n. Further, η-convertibility needs to be considered only in carrying out the first of the three kinds of actions described above. Thus, it is accounted for completely by head normalizing the two terms, comparing their binder lengths and, if these are unequal, modifying the head and arguments of the term with the shorter binder as indicated and (hypothetically) adding new bound variable arguments at the bottom of the structure in the reduction stack.

Types must be recorded with terms in order to generate the substitutions dictated by the second kind of action involved in higher-order unification. As observed, however, it suffices to remember the types of only the free variables and constants, and, therefore, low level representation of only these categories of terms needs to be modified. A feature of the substitutions that have to be considered is that they involve several arguments consisting of a free variable applied to a common list of bound variables. Our representation of application based on the Böhm tree structure has the benefit of allowing this list to be shared. We note, finally, that the unification procedure involves a genuine search: alternatives may have to be considered to substitutions chosen at certain points. The application of substitutions usually create new redexes in terms and, ultimately, to a mutation of their structures in our destructive implementation of reduction. These changes have therefore to be registered and retracted when necessary. Our implementation, that is embedded within the usual machinery for logic programming, utilizes a trail stack for this purpose.

5 Conclusion

We have presented an explicit substitution notation in this abstract and considered its use in a specific practical context: the implementation of the language λProlog. There are, obviously, several choices to be made in the course of such a use, such as the particular notation to use, whether to realize reduction destructively or nondestructively and whether to utilize the new notation merely in realizing Huet's procedure or to first lift this procedure to the changed notation as in [DHK95]. Space considerations force us to leave a detailed discussion of these tradeoffs and the rationale for our design decisions to a workshop presentation and a full paper. We also note that while educated guesses can be made regarding the best choices, the ramifications of such choices must be studied through an actual implementation. There is, at present, very little literature relating to this topic. We plan to conduct a thorough empirical study of the relevant issues after we have available a fully working implementation of λProlog.

References

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

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

[DHK95] Gilles Dowek, Thèrése Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 366–374, San Diego, California, June 1995. IEEE Computer Society Press.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[HM76] Peter Henderson and James H. Morris. A lazy evaluator. In *Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103, 1976.

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

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

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[Nad96] Gopalan Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. Technical Report TR-96-13, Department of Computer Science, University of Chicago, May 1996. To appear in *Journal of Functional and Logic Programming*.

[NM94] Gopalan Nadathur and Dale Miller. Higher-order logic programming. Technical Report CS-1994-38, Department of Computer Science, Duke University, December 1994. To appear in *Volume 5* of *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. Gabbay, C. Hogger and A. Robinson (eds.), Oxford University Press.

[NW90] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, 1990.

[NW97] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. Technical Report CS-1997-01, Department of Computer Science, University of Chicago, January 1997. To appear in *Theoretical Computer Science*.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[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.

[Per91] Fernando C. N. Pereira. Semantic interpretation as higher-order deduction. In Jan van Eijck, editor, *Logics in AI: European Workshop JELIA'90*, number 478 in Lecture Notes in Artificial Intelligence, pages 78–96, Amsterdam, Holland, 1991. Springer-Verlag, Berlin, Germany.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 153–163, 1988.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

A Term Representation and Reduction Procedures

```
/* A representation of annotated suspension expressions */

typedef struct TermCell *TermPtrType;      /* pointers to terms */

typedef struct EnvCell *EnvType;           /* environments */

typedef struct EnvCell {
      enum {DUMMY, TI_PAIR} eittag;
      int ind;
      TermPtrType tp;
      EnvType renv;
   } EnvItemType;

typedef struct {                           /* suspensions */
      TermPtrType skel;
      int ol, nl;
      EnvType env;
   } SuspType;

typedef struct {                           /* applications */
      TermPtrType fn;
      TermPtrType arg;
   } AppType;

typedef struct TermCell {                  /* terms */
      enum {OPEN, CLOSED} annot;
      enum {CONST, BV, VAR, REF, LAM, APP, SUSP} tag;
      union {
        char *name;
        int bv;
        TermPtrType ref;
        TermPtrType body;
        AppType app;
        SuspType susp;
      } val;
   } TermType;


/* The reduction stack and its TOS pointer */
TermPtrType redstack[MaxRedStack];
TermPtrType *sltop;

/* The stack for remembering applications and its TOS pointer */
TermPtrType appstack[MaxAppStack];
TermPtrType *apptop;


int numargs,         /* number of arguments in head normal form */
    numabs;          /* binder length */
```

```
/* Dereferencing a term pointer */
TermPtrType deref(TermPtrType tp) {
while (tp->tag = REF) tp = tp->val.ref; return tp;
}

/* Recording a persistent abstraction in an environment */
EnvType  add_dummy(int ind, EnvType renv) {
EnvType env;
env = (EnvType)malloc(sizeof(EnvItemType));
env->eittag = DUMMY; env->ind = ind; env->renv = renv;
return env;
}

/* Adding a real binding to an environment */
EnvType  add_binding(TermPtrType tp, int ind, EnvType renv) {
EnvType env;
env = (EnvType)malloc(sizeof(EnvItemType));
env->eittag = TI_PAIR;
env->ind = ind; env->tp = tp; env->renv = renv;
return env;
}

/* Finding the nth element in an environment */
EnvType  envnth(EnvType env, int n) {
while (n != 0) { env = env->renv; n--;}
return env;
}


/* Constructor for references */
void mkref(TermPtrType tp1, TermPtrType tp2) {
 tp1->tag = REF; tp1->val.ref = tp2;
}

/* Constructor for bound variable indices */
void mkbv(TermPtrType tp, int ind) {
 tp->tag = BV; tp->val.bv = ind;
}

/* Constructor for applications that are already annotated */
void mkapp(TermPtrType tp, TermPtrType fn, TermPtrType arg) {
 tp->tag = APP; tp->val.app.fn = fn; tp->val.app.arg = arg;
}

/* Constructor for abstractions that are already annotated */
void mklam(TermPtrType tp, TermPtrType body) {
 tp->tag = LAM; tp->val.body = body;
}

/* Constructor for open suspensions */
```

```
TermPtrType mkosusp(TermPtrType sk,int ol,int nl,EnvType env) {
TermPtrType tp;
tp = (TermPtrType)malloc(sizeof(TermType));
tp->annot = OPEN; tp->tag = SUSP;
tp->val.susp.skel = sk; tp->val.susp.ol = ol;
tp->val.susp.nl = nl; tp->val.susp.env = env;
return tp;
}

/* Modifying an already annotated suspension */
void changesusp(TermPtrType tp, TermPtrType skel,
                int ol, int nl, EnvType env) {
 tp->val.susp.skel = skel; tp->val.susp.ol = ol;
 tp->val.susp.nl = nl; tp->val.susp.env = env;
}

/* Modifying an annotated term to a suspension */
void changetosusp(TermPtrType tp, TermPtrType skel,
                  int ol, int nl, EnvType env) {
 tp->tag = SUSP; tp->val.susp.skel = skel;
 tp->val.susp.ol = ol; tp->val.susp.nl = nl; tp->val.susp.env = env;
}


/* A procedure for exposing a non-suspension structure to a term */
void lazy_read() {
TermPtrType *my_sltop;
TermPtrType tp, skp;
SuspType susp;
TermType sk;

my_sltop = sltop;

while (my_sltop >= sltop) {
  tp = *my_sltop;  susp = tp->val.susp;
  skp = deref(susp.skel); sk = *skp;

  if (sk.annot == CLOSED) /* then skeleton is not affected by substitution */
    { mkref(tp, skp);      /* make a reference for sharing */
      if (sk.tag == SUSP) *my_sltop = skp;
      else  my_sltop--;
    }

  else                     /* propagate substitution over 'open' skeleton */
    { switch (sk.tag) {

        case SUSP:         /* skeleton itself must be lazy read first */
          { my_sltop++; *my_sltop = skp; break; }

        case APP:          /* distribute suspension over arguments */
          { TermPtrType fn, arg;
            fn = deref(sk.val.app.fn); arg = deref(sk.val.app.arg);
```

```
        if (fn->annot == OPEN)
          fn = mkosusp(fn,susp.ol,susp.nl,susp.env);
        if (arg->annot == OPEN)
          arg = mkosusp(arg,susp.ol,susp.nl,susp.env);
        mkapp(tp,fn,arg); my_sltop--; break;
      }

    case LAM:          /* record the abstraction in the environment */
      { mklam(tp,mkosusp(sk.val.body,susp.ol+1,susp.nl+1,
                                add_dummy(susp.nl,susp.env)));
        my_sltop--; break;
      }

    case BV:           /* transform bound variable based on environment */
      { int ol,nl;
        ol = susp.ol; nl = susp.nl;
        if (sk.val.bv > ol)  /* renumber externally bound variable */
          { mkbv(tp, sk.val.bv - (ol - nl)); my_sltop--; }
        else         /* environment actually determines term */
          { EnvType env;
            env = envnth(susp.env,sk.val.bv);
            if (env->eittag == DUMMY)
              {         /* renumber internally bound variable */
                mkbv(tp, nl - env->ind); my_sltop--;
              }
            else         /* produce a modified version of env term */
              { skp = deref(env->tp);
                if ((nl - env->ind) == 0)
                  { mkref(tp,skp);   /* make a reference for sharing */
                    if (skp->tag == SUSP) *my_sltop = skp;
                    else my_sltop--;
                  }
                else
                  { if ((skp->annot == OPEN) && (skp->tag == SUSP))
                      {    /* simply adjust nl in the env term */
                        susp = skp->val.susp;
                        changesusp(tp,susp.skel,susp.ol,
                                   susp.nl+(nl - env->ind),susp.env);
                      }
                    else   /* embed env term in another suspension */
                      { changesusp(tp,skp,0,(nl- env->ind),NULL); }
                  }
              }
          }
        break;
      }
  }
 }
}

sltop--;
*sltop = deref(*sltop);
```

```
}       /* lazy_read */



/* A head normalization procedure */
void head_norm()  {
TermPtrType tp, app;
SuspType susp;

numargs = 0; numabs = 0; apptop = appstack;

while (1) {
 tp = *sltop;
 switch (tp->tag) {
   case VAR: case BV:       /* head normal form has been found */
   case CONST:   return;

   case SUSP:                /* expose a non suspension structure */
      *(++sltop) = tp; lazy_read(); break;

   case APP:                 /* stack app and args, look for redex */
     *apptop++ = tp;
     *sltop++ = deref(tp->val.app.arg);
     *sltop = deref(tp->val.app.fn);
     numargs++; break;

   case LAM:                 /* contract if redex, else descend into body */
     { tp = deref(tp->val.body);
       if (numargs == 0)   /* there is no redex */
         { *sltop = tp; numabs++; }
       else                 /* head redex found; contract it */
         { app = *apptop--;
           if ((tp->annot == OPEN) && (tp->tag == SUSP) &&
                   ((susp = tp->val.susp).ol > 0) &&
                    (susp.env->eittag == DUMMY) &&
                    (susp.nl == (susp.env->ind + 1)))
             { changetosusp(app,susp.skel,susp.ol,susp.nl,
                            add_binding(*(sltop-1),susp.nl-1,
                                        susp.env->renv)); }
           else
             { changetosusp(app,tp,1,0,
                            add_binding(*(sltop-1),0,NULL));}
           *(--sltop) = app; numargs--;
         }
       break;
     }
 }
}
}  /* head_norm */
```