EFFICIENT UNIFICATION OF QUANTIFIED TERMS*

JOHN STAPLES† AND PETER J. ROBINSON

▷ Conventional logic-programming languages rely fundamentally on symbolic computation with quantifier-free terms. Much theoretical logic uses the richer vocabulary of quantified terms, however. In this paper we sketch some first steps in a program of research for developing data structures and algorithms to support efficient computation directly on quantified terms. We describe a simple concept of quantified term, and efficient unification algorithms for both structure-sharing and non-structure-sharing representations of those terms. The efficiency of the approach results from the techniques used to represent terms, which enable naive substitution to implement correct substitution for quantified terms. The non-structure-sharing unification algorithm described here has been prototyped by modification of a conventional logic-programming interpreter. ◁

1. INTRODUCTION

The use of quantifiers is widespread in informal human reasoning and computation. For example, the logical quantifiers "for all" and "there exists" are well known, and the lambda calculus relies almost exclusively on the single quantifier λ for its expressive power. Modes of expression such as the following, where x is a variable and $\cdots$ is some term, are also naturally formalized by quantifiers:

The least x such that $\cdots$

The set of x such that $\cdots$

The sequence with xth term $\cdots$

The integral with respect to x of $\cdots$

The procedure with formal parameter x and body $\cdots$

*The work reported here was supported in part by the Australian Research Grants Scheme.
†The first author thanks Alan Bundy for hospitality at the Department of Artificial Intelligence, University of Edinburgh, during the preparation of this paper.
Address correspondence to Dr. J. Staples, Department of Computer Science, University of Queensland, Australia 4067.
Received December 1986; accepted July 1987.

52 Vanderbilt Ave., New York, NY 10017 0743-1066/88/$3.50

For simplicity we consider the following simple syntax for quantified terms, in which each quantifier binds exactly one variable.

1.1. Terms

Our definitions assume that the following primitive syntactic categories have already been fixed: an infinite set of variables; for each n a set of n-place function symbols; and a fixed set of quantifiers. These categories are assumed to be disjoint, except that the sets of function symbols may overlap.

Definition 1.1.1. We define conventional ("old") terms recursively as follows:

(a) Each variable is a term. We use lowercase letters $x, y, \ldots,$ to denote variables.

(b) For each n-place function symbol f and each n-tuple $t_1, \ldots, t_n$ of terms, $f(t_1, \ldots, t_n)$ is a term. By convention we regard this definition as prescribing that each 0-place function symbol is a term.

(c) For each quantifier Q, each variable x, and each term t, Qxt is a term.

Next we introduce the concept of unification which is appropriate for our quantified terms.

1.2. Unification

A unifier for two terms t and u is a substitution σ such that $\sigma t = \sigma u$. For quantified terms we naturally use the correct concept of substitution, which substitutes only for free occurrences of variables and which avoids capture of variables. For example, substituting u for y in $a(y, Qyy)$ results in $a(u, Qyy)$, since only free occurrences of y are replaced in the substitution. Also, substituting x for y in $Qxa(y, x)$ results in, say, $Qza(x, z)$; the name of the bound variable x has been changed to z to avoid the result $Qxa(x, x)$, which does not express the intended semantics of substitution and of which it is said that the intended free occurrence of x has been *captured* by the quantifier.

This unification concept depends on a notion of equality of terms. The elementary unification algorithm for free-variable terms uses, as its equality, syntactic identity of free-variable terms. That is not appropriate for quantified terms, since one wishes to regard the choices of names of bound variables as insignificant. For unification we consider as equal, terms which differ only in the names of bound variables. In contexts such as the lambda calculus this relation is called α-equivalence; we use the same name here. The following examples illustrate the intended unification concept. Here Q denotes some quantifier, x, y are variables, a is a function symbol, and c is a constant.

(1) $Qxa(x)$ and $Qya(y)$ do unify, by the identity substitution, since the two differ only by a change of bound variable. This illustrates that bound variables cannot be treated by a unification algorithm in the same way as constants.

(2) $Qxa(x)$ and $Qya(c)$ do not unify. This illustrates that bound variables cannot be treated in the same way as free variables.

(3) $QxQxa(x, x)$ and $QxQya(x, y)$ do not unify, illustrating that the meaning of a bound occurrence of a variable is local to the scope of its quantifier.

(4) $Qxa(y)$ and $Qya(x)$ do unify, but not to $Qxa(x)$ or $Qya(y)$. The result could be described as $Qza(x)$.

Thus a unification algorithm for quantified terms has to manage bound variables and their scopes, and has to perform correct substitutions. For the sake of efficiency however we would prefer to use naive substitution, as in free-variable unification. Thus we introduce new representations of quantified terms which allow naive substitution to represent correct substitution. That is described in the following sections, for both structure-sharing and non-structure-sharing versions of the term representation.

2. A NON-STRUCTURE-SHARING REPRESENTATION OF QUANTIFIED TERMS

The key to the efficiency of our unification algorithm for quantified terms is to use a representation of terms which makes naive substitution correct. Then, implementation of a variable instantiation requires only a pointer update, as for free-variable terms.

To achieve that, we recognize variable dereferencing in our theoretical discussion of terms, and integrate it with the management of quantifiers. Consequently we need to consider an equivalence relation on terms which expresses the role of variable dereferencing. That is not an extra burden, since an equivalence relation is required in any case to express the significance of changes of bound variable.

This section summarizes work developed in [4]; see there for proofs which are omitted here.

2.1. New Terms

The definition of non-structure-sharing ("new") terms introduced in this section is similar to the definition of old terms in Section 1.1. The only differences are that a new lexical primitive *nil* is introduced and Definition 1.1.1(a) is changed to the following which provides a notation for instantiated variables:

(a) For each variable x and each term t, $x(nil)$ and $x(t)$ are terms.

2.2. Translating from New to Old Terms

We describe the intention of our new term notation by defining a translation from new terms to old. To simplify our discussion we assume that an infinite sequence of variables is available which does not include any variables occurring free in the formula being translated. For simplicity we assume that this sequence is fixed throughout. We also assume that our new representation of terms does not use those variables at all.

Definition 2.2.1. We use $s.1, s.2, \ldots$ to denote the variables in this special sequence.

We need to consider subterms when defining and reasoning about the unification algorithm. Indeed, that requirement arises throughout the section, starting with the mapping from new terms to old which we are about to define. The following definitions will be widely used.

Definition 2.2.2. We define a binding state to be a one-one function from (finitely many) variables to positive integers. We shall often write, for a binding state B and a variable x, $B(x) = \perp$ to denote that x is not in the domain of B. We write $\varnothing$ for the binding state with empty domain.

It is convenient to say that a positive integer n is above a binding state B if for all m in the range of B, $m < n$. Note that all positive integers are above $\varnothing$.

We write $\nu(B)$ for the least positive integer above B. We also write $\nu(L, R)$ for the least positive integer which is above both the binding states L and R. We say that positive integers $n \geq \nu(B)$ are above B.

We shall make frequent use of an update operation $[n/x]$ on binding states, as follows. For each binding state B, positive integer $n \geq \nu(B)$, and variable x, $[n/x]B$ denotes the binding state such that

(a) $([n/x]B)(x) = n$;

(b) for $y \neq x$, $([n/x]B)(y) = B(y)$.

We write $[x]B$ for $[\nu(B)/x]B$. We also write, for a pair L, R of binding states, $[n/x, y](L, R)$ for $([n/x]L, [n/y]R)$ and $[x, y](L, R)$ for $[\nu(L, R)/x, y](L, R)$. We abbreviate $[x, x](L, R)$ to $[x](L, R)$.

Definition 2.2.3. Our definition of a mapping O from new terms to old is recursive. For the sake of the recursive definition we define more generally, for each binding state B, a mapping O_B. Then we finally define O to be $O_\varnothing$.

The clauses of the recursive definition of O_B are as follows:

(a) If $B(x) = \perp$ then $O_B(x(nil)) = x$.

(b) If $B(x) = \perp$ and $t \neq nil$ then $O_B(x(t)) = O_\varnothing(t)$. ("Dereferencing escapes bindings.")

(c) If $B(x) \neq \perp$ then for all terms t, $O_B(x(t)) = s.B(x)$. ("Binding takes precedence over dereferencing.")

(d) For all $m \geq 0$, all m-place function symbols f, and all terms $t_1, \ldots, t_m$,

$$O_B(f(t_1, \ldots, t_m)) = f(O_B(t_1), \ldots, O_B(t_m)).$$

(e) For all quantifiers Q, variables x, and terms t,

$$O_B(Qxt) = Qs.\nu(B)O_{[x]B}(t).$$

2.3. Substitutions

Definition 2.3.1. The definition of substitution for our new terms is simpler than for old terms. A preliminary definition of free occurrences is not required. Substitu-

tion of the new term t for x in the new term u is just naive substitution of t for *nil* in every occurrence of $x(nil)$ in u. In practice it is natural to implement all occurrences of a variable x in a new term by pointers to a single occurrence of $x(\cdot)$. Then, instantiating x can be achieved by changing the single occurrence of *nil*. To avoid confusion we use the notation $\langle t/x\rangle u$ to denote the result of new substitution. It will be convenient for later discussion to state this definition recursively as follows. The extension to parallel substitution for several variables is straightforward and is omitted.

(a) $\langle t/x\rangle x(nil) = x(t)$.

(b) For every variable $y \neq x$, $\langle t/x\rangle y(nil) = y(nil)$.

(c) For every variable z and $u \neq nil$, $\langle t/x\rangle z(u) = z(\langle t/x\rangle u)$.

(d) $\langle t/x\rangle f(t_1, \ldots, t_n) = f(\langle t/x\rangle t_1, \ldots, \langle t/x\rangle t_n)$.

(e) $\langle t/x\rangle Qzu = Qz\langle t/x\rangle u$, whether or not $x = z$, and regardless of occurrences of z in t.

The correctness of our approach to substitution can be supported as follows.

Proposition 4.1. For all new terms t and u and variables x,

$$O(\langle t/x\rangle u) = [O(t)/x]O(u).$$

2.4. Equivalence of New Terms

Here we introduce an equivalence relation on terms which expresses the fact that changes of bound variable and dereferencing are insignificant. We follow convention in calling it α-equivalence. Just as the mapping O from new terms to old was generalized in Section 2.2 for the sake of the recursive definition, so we need to generalize the concept of α-equivalence.

Since α-equivalence relates two terms, its generalization considers a pair of subterms together with a corresponding pair of binding states. In fact, we define a relation $\sim_{L,R}$ for every pair L, R of binding states and every positive integer n which is above both L and R.

For theoretical convenience the definition of $\sim_{L,R}$ is apparently nonconstructive, but there is no difficulty in giving an equivalent constructive form.

Definition 2.4.1. The relation $\sim_{L,R}$ is defined recursively. For arbitrary terms t and u, $t \sim_{L,R} u$ if and only if one of the following conditions is satisfied:

(a) $t = x(v)$, $u = y(w)$ and $L(x) = R(y) \neq \perp$.

(b) $t = x(v)$, $L(x) = \perp$, $v \neq nil$, and $v \sim_{\varnothing,R} u$.

(c) $u = y(w)$, $R(y) = \perp$, $w \neq nil$, and $t \sim_{L,\varnothing} w$.

(d) $t = x(nil)$, $L(x) = R(x) = \perp$, and $u = t$.

(e) $t = f(t_1, \ldots, t_m)$, $u = f(u_1, \ldots, u_m)$, and $t_i \sim_{L,R} u_i$, $i = 1, \ldots, m$.

(f) $t = Qxv$, $u = Qyw$, and there is a positive integer k above both L and R such that for all $n \geq k$, $v \sim_{[n/x,y](L,R)} w$.

We shall abbreviate "there is a positive integer k above both L and R such that for all $n \geq k$" to "for all n sufficiently large."

Definition 2.4.2. For arbitrary terms t and u, $t \sim_\alpha u$ if and only if $t \sim_{\varnothing,\varnothing} u$.

It is not immediately clear that these equivalence relations are computable, since Definition 2.4.1(f) refers to an infinite number of values of n. The form of the definition stated is the most convenient one from which to begin reasoning about the relations, but it is in fact equivalent to require $n = k = v(L, R)$, so that computability is not in doubt. These facts are proved in [4].

2.5. *A Unification Algorithm*

Our algorithm is similar in structure to the unification algorithms for free-variable formulas which are used in conventional PROLOG interpreters. Its performance on free-variable terms is essentially the same as the free-variable algorithm's. This algorithm has been prototyped by modification of the York Portable PROLOG Interpreter [2]. Unification of quantified terms requires a second form of occurs check. The check is that an object variable must not be instantiated to a value which includes bound occurrences of variables.

For convenience in our discussion of the cases when the algorithm detects that unification is impossible, our algorithm outputs a sequence $o_1, \ldots, o_k$ of objects, each of which is either a substitution or *fail*. The intention is that if any *fail* occurs in an output sequence, the input is not unifiable. Otherwise, the composition $o_k \circ \cdots \circ o_1$ of the output sequence of substitutions is a most general unifier of the input sequence.

First we sketch a simple example of using the unification algorithm. Consider the problem of unifying the two terms which are conventionally denoted

$$a(x, Qxb(x, z)) \quad \text{and} \quad a(y, Qyb(y, x)),$$

where Q is a quantifier, a, b are function symbols and x, y, and z are variables. In the notation introduced in Definition 1.1.1, x, y, and z are denoted $x(nil)$, $y(nil)$, and $z(nil)$ respectively, so that the above terms are denoted

$$a(x(nil), Qxb(x(nil), z(nil))) \quad \text{and} \quad a(y(nil), Qyb(y(nil), x(nil))).$$

For readability in this example we abbreviate $x(nil)$, $y(nil)$ and $z(nil)$ to x, y, and z respectively, so that initially our abbreviated notation is the same as the conventional notation.

Essentially, the unification algorithm described in Definition 2.5.1 below unifies a sequence of pairs of terms, just as conventional free-variable unification does. Our algorithm also associates with each pair of terms in the sequence a pair of binding states. As the example will indicate, the left binding state records which variables in the left term are bound global to the term; similarly for the right binding state.

We call a pair of terms, subscripted by a pair of binding states, an *item*. To be more precise, our unification algorithm unifies sequences of items.

The input to the unification algorithm in our example is the item

$$a(x, Qxb(x, z)), a(y, Qyb(y, x)))_{\varnothing,\varnothing}.[\,].$$

Our algorithm begins to unify this sequence in the same way as the familiar free-variable unification algorithm. It matches the outer function symbols of the first pair and generates two pairs corresponding to the two places of the function symbol. The result is to simplify the initial problem to

$$(x, y)_{\varnothing,\varnothing}.(Qxb(x, z), Qyb(y, x))_{\varnothing,\varnothing}.[\,].$$

In accordance with the principle stated in Definition 2.2.3, that binding takes precedence, the next step first checks whether x has a binding value in $\varnothing$, and similarly for y. As neither does, the possibility of dereferencing x or y is considered, but neither can be dereferenced. Thus the situation is handled as in free-variable unification. The instantiation of x to y is made part of the output substitution, and (naive) substitution of y for x is applied to the remainder of the problem. The success of this step is conditional on an occurs check, as in free-variable unification, and also on a second occurs check which ensures that there are no bound occurrences of variables in the subterms which will become the value of x. The result is as follows [note that our notation begins to depart from the conventional, as we record the instantiation of x to y by $x(y)$]:

$$(Qxb(x(y), z), Qyb(y, x(y)))_{\varnothing,\varnothing}.[\,].$$

Next the quantifiers are matched and the bodies they quantify are taken as a pair of terms in a simplified problem. Also, the fact that x is bound in the left side and y is the corresponding bound variable in the right side is recorded by associating binding states

$$L = \{x = 1\} \quad \text{and} \quad R = \{y = 1\}.$$

Thus the simplified problem is

$$(b(x(y), z), b(y, x(y)))_{L,R}.[\,].$$

Since a function symbol is now at the top level, we again simplify as in free-variable unification, without changing the binding states. That results in the simplified problem

$$(x(y), y)_{L,R}.(z, x(y))_{L,R}.[\,].$$

For each of x and y the algorithm first consults the associated binding state. As x has the same value in L as does y in R, unification of this pair succeeds (without generating any increment to the output substitution). The remaining problem is then to unify

$$(z, x(y))_{L,R}.[\,].$$

As the terms are again both variables, the associated binding states are again consulted. On this occasion neither has a binding value. The next step is to see if either variable can be dereferenced. As x has been instantiated to y, dereferencing of x is the next step. Recall from Definition 2.2.3 that dereferencing escapes the scope of all quantifiers. That is implemented by changing R to $\varnothing$ for the simplified pair:

$$(z, y)_{L,\varnothing}.[\,].$$

As z has no value in L and y has none in $\varnothing$, and as neither can be dereferenced, the instantiation of z to y is generated as the next and final part of the output

substitution. Also this substitution is applied to the remainder of the sequence, though in this case there is nothing to do.

An empty sequence of term pairs has now been reached, signaling successful termination of the algorithm. The composition of the output substitutions is the overall unifying substitution generated by the algorithm. Note that this unifying substitution, when applied to the terms required to be unified, does not produce identical terms. In fact it produces

$$a(x(y), Qxb(x(y), z(y))) \quad \text{and} \quad a(y, Qyb(y, x(y))).$$

These terms are however equivalent modulo changes of bound variable and dereferencing. That is, they are α-equivalent in the sense defined in Definition 2.4.2.

Our unification algorithm can be described as follows. We define an *item* as a pair $(t, u)_{L,R}$ of terms subscripted by a pair of binding states.

Definition 2.5.1. The function unify, which maps sequences of items to sequences $o_1, \ldots, o_k$ as described above, is defined by defining, for all sequences T of items, unify(T) recursively as follows ([ ] denotes the empty sequence):

(a) unify ([ ]) = [*id*].

(b) Otherwise, if $t = x(v)$ and $u = y(w)$ and $L(x) = R(y) \neq \perp$, then
$$\text{unify}((t, u)_{L,R}.T) = \text{unify}(T).$$

(c) Otherwise, if $t = x(v)$, $v \neq nil$, and $L(x) = \perp$, then
$$\text{unify}((t, u)_{L,R}.T) = \text{unify}((v, u)_{\varnothing,R}.T).$$

(d) Otherwise, if $u = y(w)$, $w \neq nil$, and $R(y) = \perp$, then
$$\text{unify}((t, u)_{L,R}.T) = \text{unify}((t, w)_{L,\varnothing}.T).$$

(e) Otherwise, if $t = u = x(nil)$ and $L(x) = R(x) = \perp$, then
$$\text{unify}((t, u)_{L,R}.T) = \text{unify}(T).$$

(f) Otherwise, if $t = x(nil)$ and $L(x) = \perp$ and u has no R-free occurrence of x and no R-bound occurrence of any variable, then
$$\text{unify}((t, u)_{L,R}.T) = \langle u/x \rangle.\text{unify}(\langle u/x \rangle T).$$

(g) Otherwise, if $u = y(nil)$ and $L(y) = \perp$ and t has no L-free occurrence of y and no L-bound occurrence of any variable, then
$$\text{unify}((t, u)_{L,R}.T) = \langle t/y \rangle.\text{unify}(\langle t/y \rangle T).$$

(h) Otherwise, if $T = f(t_1, \ldots, t_n)$ and $u = f(u_1, \ldots, u_n)$, then
$$\text{unify}((t, u)_{L,R}.T) = \text{unify}((t_1, u_1)_{L,R}.\cdots.(t_n, u_n)_{L,R}.T).$$

(i) Otherwise, if $t = Qxv$ and $u = Qyw$, then
$$\text{unify}((t, u)_{L,R}.T) = \text{unify}((v, w)_{[x,y](L,R)}.T).$$

(j) Otherwise
$$\text{unify}((t, u)_{L,R}.T) = [\mathit{fail}].$$

A proof of correctness of this algorithm is given in [4].

3. STRUCTURE SHARING FOR QUANTIFIED TERMS

Structure sharing refers to the representation of a term t as a pair (e, s), where s is a *skeleton* which intuitively represents the top levels of the terms t, and which intuitively can be converted to t by performing on s a substitution defined by e. The *environment* e is in our approach not explicitly a substitution, since it also encodes information about the bindings of variables in which t appears as a subterm. The sharing referred to is the sharing of the top levels of t, which is achieved by implementations in which several terms share the same skeleton. In practice, efficient use of the structure sharing described here requires sharing in the representation of environments also.

In this section we refer to our structure-sharing terms as new terms. They are not to be confused with the non-structure-sharing terms which were described in Section 2 as new terms. However, the management of bound variables for our structure-sharing terms is similar to the previous case, and so is the unification algorithm. Proofs omitted here are given in [3].

3.1. Basic Definitions

The basic lexical objects from which our terms are constructed are as for the conventional terms described in Section 1.1, with the following addition.

Definition 3.1.1. We use an infinite set of formal values. They play essentially the same role as variable values in the binding states of the non-structure-sharing approach. Individual formal values are usually denoted $v, v', \ldots$.

Definition 3.1.2. Skeletons are just conventional quantified terms. Formal values do not appear in skeletons.

We apply the old concepts of free and bound occurrences of variables to skeletons. That should not be confused with the concepts of free and bound variable occurrences to be defined in Section 3.4.

The definitions of environment value, block, environment, and term are mutually recursive, as follows. For readability we state the several sections of the definition as if they were separate definitions.

Definition 3.1.3. As environment value is defined to be either a formal value or a term.

Definition 3.1.4. A block is a function whose domain is some finite set of variables, and each of whose values is an environment value. We write $\varnothing$ for the block with empty domain. We may write $b(x) = \perp$ to denote that the variable x is not in the domain of the block b. To state explicitly that the domain of a block b comprises the distinct variables $x_1, \ldots, x_n$ and that their values in b are $v_1, \ldots, v_n$ respectively, we may write $(v_1, \ldots, v_n/x_1, \ldots, x_n)$ for b.

Definition 3.1.5. An environment is a finite sequence $b_1 \ldots . b_n$, $n \geq 0$, of blocks. The infix operator "." will be used both as a separator between elements of sequences and as a concatenation operator on finite sequences. We write [ ] for the empty sequence.

Definition 3.1.6. A term is a pair (e, s), where e is an environment and s is a skeleton. It is convenient to extend the use of the operator "." as follows. For terms $t = (e, s)$ and environments e', $e'.t$ denotes $(e'.e, s)$ and $t.e'$ denotes $(e.e', s)$.

That ends the clauses of the mutually recursive definitions.

A basic purpose of an environment e is to define a block called the display of e, as follows.

Definition 3.1.7. The display $display_e$ defined by an environment e is a block which is defined recursively as follows:

(a) $display_{[]} = \varnothing$.

(b) For environments $e = b.e'$:

(ba) If $b(x) = \perp$ then $display_e(x) = display_{e'}(x)$.

(bb) If $b(x)$ is a formal value, then $display_e(x) = b(x)$.

(bc) If $b(x)$ is a term, then in the notation of Definition 3.1.6, $display_e(x) = b(x).e'$.

We shall often confuse e and $display_e$, provided the context can resolve the ambiguity. In particular we often write $e(x)$ instead of $display_e(x)$.

3.2. *Entries and Substitutions*

An entry is essentially an extension of the non-structure-sharing concept of updating a binding state. The name implies entry into a new, local block. Here we state the definitions of entries and substitutions.

Definition 3.2.1. The entry $[b]$ defined by a block b is a function from terms to terms defined (in prefix operator notation) by $[b]t = b.t$. When it is desired to describe an entry $[b]$ by an explicit enumeration of the variables $x_1, \ldots, x_n$ in the domain of b and their respective values $v_1, \ldots, v_n$, we may denote the entry $[v_1, \ldots, v_n/x_1, \ldots, x_n]$.

We also apply entries to environments, again using prefix operator notation. For environments e, $[b]e = b.e$.

Definition 3.2.2. The substitution $\langle b \rangle$ defined by a block b is a function from terms to terms defined (in prefix operator notation) by $\langle b \rangle t = t.b$. When it is desired to describe a substitution $\langle b \rangle$ by an explicit enumeration of the variables $x_1, \ldots, x_n$ in the domain of b and their respective values $v_1, \ldots, v_n$, we may denote the substitution by $\langle v_1, \ldots, v_n/x_1, \ldots, x_n \rangle$.

We also apply substitutions to environments, again using prefix operator notation. For environments e, $\langle b \rangle e = b.e$.

3.3. *Correctness of Substitutions*

In this section we define translations O from our new terms and new substitutions to old terms and old substitutions respectively. We then note that our new substitution representation is correct, in the sense that substitution commutes with

translation to old. We follow tradition by confusing old terms which differ only by changes of bound variable. On the other hand, no such liberties are taken with new terms.

First the translation of terms. We assume that the formal values used in new terms represent variables in old terms.

Definition 3.3.1. For all terms $t = (e, s)$,

(a) If s is a variable and $e(s)$ is undefined, then $O(t) = s$.

(b) If s is a variable and $e(s)$ is a formal value, then $O(t) = e(s)$.

(c) If s is a variable and $e(s) = (e', s')$ is a term, then $O(t) = O(e(s))$.

(d) If $s = f(s_1, \ldots, s_n)$ then $O(t) = f(O(e, s_1), \ldots, O(e, s_n))$.

(e) If $s = Qxs'$ then $O(t) = QvO(e', s')$, where $e' = [v/x].e$, and where v is some formal value not used in e.

We translate new substitutions to old componentwise, as follows. Note that, as for terms, both new variables and formal values are translated to old variables, without changing the notation.

Definition 3.3.2. For all new substitutions $\sigma = \langle b \rangle$, $O(\sigma)$ is the old substitution which instantiates old variables x as follows:

(a) If $b(x) = \perp$, then $O(\sigma)$ does not instantiate x.

(b) If $b(x)$ is a formal value, then $O(\sigma)$ instantiates x to $b(x)$.

(c) If $b(x)$ is a term, then $O(\sigma)$ instantiates x to $O(b(x))$.

Proposition (Correctness of the substitution representation). For all new terms t and new substitutions σ, $O(\sigma t) = O(\sigma)O(t)$.

3.4. Free and Bound Occurrences of Variables

For our new terms we classify variable occurrences in one of four ways. Each of these ways will be called a role. Each variable occurrence in a term has at most one role in that term. It is possible for a variable occurrence to have no role. That occurs when the occurrence is within a redundant environment value.

We call the variable roles free, globally bound, locally bound, and instantiated. The free and locally bound roles correspond to the classical free and bound roles for variable occurrences in old terms. Intuitively, globally bound occurrences in a term t are those whose dereferencing causes a formal value to appear in $O(t)$. Instantiated occurrences are those whose dereferencing causes one or more subterms to appear in $O(t)$.

We make the above intuitions precise by means of the following recursive definition. We also include a definition of the formal value of each globally bound variable occurrence. Note that our definition relies on the old concept of free occurrence of variable, for the discussion of variable occurrences in skeletons.

Definition 3.4.1. For terms $t = (e, s)$, we define the free, globally bound, locally bound, and instantiated occurrences of variables in t, and the formal values to which globally bound variable occurrences are bound, by mutual recursion as follows:

(a) If $s = x$, a variable, and $e(x) = \perp$, then the skeletal occurrence of x is free. No other variable occurrences in t have any role.

(b) If $s = x$ and $e(x)$ is a formal value, then the skeletal occurrence of x is globally bound to $e(x)$. No other variable occurrences in t have any role.

(c) If $s = x$ and $e(x)$ is a term, then the skeletal occurrence of x is an instantiated variable occurrence. Each variable occurrence which has a role in $e(x)$ has the same role in t, and the formal values of globally bound occurrences are the same in t as in $e(x)$.

(d) If $s = f(s_1, \ldots, s_n)$, then each variable occurrence in each s_i which has a role in s_i has the same role, and formal value if applicable, in t.

(e) If $s = Qxs'$, then all free occurrences of x in s' [that is, free in the old sense, regardless of their role in (e, s')] are locally bound occurrences in t. All free occurrences of other variables in s' have the same role in t as in (e, s'). Also, if y occurs free (in the old sense) in s' and $e(y)$ is a term, then all variable occurrences with a role in $e(y)$ have the same role in t. All globally bound occurrences in t have the same formal value as in (e, s').

3.5. *Equivalence of Terms*

For our structure-sharing new terms there are four reasons for equivalence between terms: changes of bound variable, dereferencing of skeleton variables, differing block structures for environments, and the possible inclusion in blocks of redundant instantiations. We define an equivalence relation $\sim$ on terms recursively as follows. For theoretical convenience the definition is nonconstructive, but equivalent constructive versions are easily given.

Definition 3.5.1. For arbitrary terms $t = (e, s)$ and $t' = (e', s')$ we define when the relation $t \sim t'$ holds, by recursion following the wellorderings of t and t'. At each stage the relation holds provided one of the following cases is satisfied:

(a) $s = x$, $s' = x'$ and $e(x) = e'(x')$ is a formal value.

(b) $s = x$, $e(x)$ is a term, and $e(x) \sim t'$.

(c) $s' = x'$, $e'(x')$ is a term, and $t \sim e'(x')$.

(d) $s = x = s'$ and $e(x) = e'(x) = \perp$.

(e) $s = f(s_1, \ldots, s_n)$ and $s' = f(s'_1, \ldots, s'_n)$ and $(e, s_i) \sim (e', s'_i)$, $i = 1, \ldots, n$.

(f) $s = Qxr$, $s' = Qx'r'$, and for all but finitely many formal values v, $([v/x]e, s) \sim ([v/x']e', s')$.

It will be convenient to abbreviate "all but finitely many" to "cofinitely many".

3.6. *Unifiers*

In this structure sharing context the appropriate concept of item is simply a pair of terms. That is, in general we unify a sequence of pairs of terms.

Although our structure-sharing representation of terms provides for globally bound variable occurrences, to obtain a concept of unification comparable to that of Section 2 we exclude globally bound occurrences from unifying substitutions.

Definition 3.6.1. A substitution $\sigma = \langle b \rangle$ is a unifier of an item (t, u) if and only if $\sigma(t) \sim \sigma(u)$ and no values of b have any globally bound variable occurrences.

On this basis, concepts such as most general unifier are developed as in the non-structure-sharing case.

3.7. *A Unification Algorithm*

The algorithm is a straightforward adaptation of the non structure sharing algorithm. In this case an item (t, t') is simply a pair of terms.

To help comparison with the non-structure-sharing algorithm we first give a structure-sharing version of the worked example of Section 2.5. Recall that the problem is to unify the terms conventionally denoted

$$a(x, Qxb(x, z)) \quad \text{and} \quad a(y, Qyb(y, x)).$$

For ease of comparison with the non-structure-sharing case we assume that the initial structure sharing representations have those conventional terms as skeletons and have empty environments. Hence we unify the following one-item sequence:

$$[([\,], a(x, Qxb(x, z))), ([\,], a(y, Qyb(y, x)))].$$

The first step is to reduce to the arguments of a. Hence we unify

$$[([\,], x), ([\,], y)].[([\,], Qxb(x, z)), ([\,], Qyb(y, x))].$$

The algorithm deals with the first of these items by instantiating x to $([\,], y)$, which we denote y'. It then remains to consider

$$[((y'/x), Qxb(x, z)), ((y'/x), Qyb(y, x))].$$

To deal with the quantifiers, the algorithm chooses a new formal value v and makes entries $[v/x], [v/y]$ in the left and right environments respectively. Writing $e_1 = (v/x).(y'/x)$ and $e_2 = (v/y).(y'/x)$, it remains to unify

$$[(e_1, b(x, z)), (e_2, b(y, x))].$$

The next simplification is to the unification of

$$[(e_1, x), (e_2, y)].[(e_1, z), (e_2, x)].$$

As $e_1(x) = e_2(y)$ is a formal value, the first item is removed, leaving the problem of unifying

$$[(e_1, z), (e_2, x)].$$

Here neither z nor x is globally bound, but $e_2(x) = y'$, so the algorithm simplifies the problem to the unification of

$$[(e_1, z), y'].$$

Instantiation of z to y' completes the unification. The unified terms, which are equivalent in the sense of Definition 3.5.1, are

$((y'/x).(y'/z), a(x, Qxb(x, z)))$,

$((y'/x).(y'/z), a(y, Qyb(y, x)))$.

Definition 3.7.1. The function unify, which maps sequences of items to sequences $o_1, \ldots, o_k$ as described above, is defined by defining, for all sequences T of items, unify(T) recursively as follows ([ ] denotes the empty sequence):,

(a) unify([ ]) = [*id*].

For terms $t = (e, s)$ and $t' = (e', s')$ and sequences of items T,

(b) If $s = x$, $s' = x'$ and $e(x) = e'(x')$ is a formal value, then

$$\text{unify}((t, t').T) = \text{unify}(T).$$

(c) Otherwise, if $s = x$ and $e(x)$ is a term, then

$$\text{unify}((t, t').T) = \text{unify}((e(x), t').T).$$

(d) Otherwise, if $s' = x'$ and $e'(x')$ is a term, then

$$\text{unify}((t, t').T) = \text{unify}((t, e'(x')).T).$$

(e) Otherwise, if $s = x = s'$ and $e(x) = e'(x) = \perp$, then

$$\text{unify}((t, t').T) = \text{unify}(T).$$

(f) Otherwise, if $s = x$, $e(x) = \perp$ and t' has no free occurrence of x and no globally bound occurrence of any variable, then

$$\text{unify}((t, t').T) = \langle t'/x \rangle.\text{unify}(\langle t'/x \rangle T).$$

(g) Otherwise, if $s' = x'$, $e'(x') = \perp$, and t has no free occurrence of x and no globally bound occurrence of any variable, then

$$\text{unify}((t, t').T) = \langle t/x' \rangle.\text{unify}(\langle t/x' \rangle T).$$

(h) Otherwise, if $s = f(s_1, \ldots, s_n)$ and $s' = f(s'_1, \ldots, s'_n)$, then

$$\text{unify}((t, t').T) = \text{unify}(((e, s_1), (e', s'_1)). \cdots .((e, s_n), (e', s'_n)).T).$$

(i) Otherwise, if $s = Qxr$ and $s' = Qx'r'$, then, for an arbitrarily chosen formal value v which is independent of $(t, t').T$,

$$\text{unify}((t, t').T) = \text{unify}((([v/x]e, r), ([v/x']e', r')).T).$$

(j) Otherwise

$$\text{unify}((t, t').T) = [\mathit{fail}].$$

A specification and proof of correctness for this algorithm are detailed in [3].

4. DIRECTIONS FOR FURTHER WORK

The work described above is the beginning of a study of efficient computation with quantified terms. We sketch some further directions for our program of research by two simple examples. Our examples are based on the well-known vocabulary of

the lambda calculus. The lambda calculus is however just an example; it is not a prerequisite to or foundation for our work. We could have chosen instead a vocabulary suitable for, say, a symbolic integration calculus, or a relational calculus.

We recall that informally, lambda terms are built from variables, using a binary operator $(\cdot,\cdot)$ and a quantifier λ as follows:

(1) Each variable is a lambda term.

(2) For all lambda terms A and B, (AB) is a lambda term.

(3) For all lambda terms A and all variables X, (λXA) is a lambda term.

We are interested in supporting logical reasoning both by logic programming languages and by more general tools for reasoning such as proof editors. For simplicity, however, our examples both concern logic programming.

4.1. Example: Defining Lambda Terms

How might an extended logic programming language state a definition of the set of terms of the lambda calculus? We suggest that the following PROLOG-like notation is reasonably natural, and that its natural semantics captures the informal definition. We assume that the operator notation used in the above informal description of lambda terms is available, and that λ is a legal symbol. We set

lterm(A) :- *object-variable*(A).

lterm$((AB))$:- *lterm*(A), *lterm*(B).

lterm$((\lambda XA))$:- *lterm*(A).

Here are some indications of the semantics intended for this example.

(1) Two sorts of variable are in use. Those denoted $A, B, \ldots$ we call meta variables. Technically, substitution for meta variables is naive in the sense that all occurrences of meta variables are free occurrences and substitution may capture variables. For example, unification of λXA with λXX succeeds and instantiates A to X. The variables denoted $X, Y, \ldots$ we call object variables. Substitution for object variables, as usual for quantified terms, is nonnaive in that substitution acts only on free occurrences of variables and avoids capture of variables. For example, for distinct object variables X and Y, instantiation of Y to X in (λXY) does not give (λXX), since that would involve the quantifier capturing the instantiating occurrence of X. Instead, instantiation of Y to X in (λXY) results in a term such as (λZX). The choice of Z is not important, since quantified terms which differ only by changes of bound variable are regarded as equivalent.

(2) The language supports the declaration of quantifiers, each of which binds an object variable. The example assumes λ has been declared a quantifier. Unification unifies terms up to the equivalence which is defined by allowing changes of bound variables. For example, (λXX) unifies with (λYY), since these terms differ only by a change of bound variable. On the other hand, neither unifies with (λXc), for any constant c, since bound occurrences of variables may only be changed, not instantiated. Note that $(X(\lambda XX))$ unifies

with $(c(\lambda XX))$, illustrating that the meaning of a bound occurrence of a variable is local to the scope of its binding quantifier.

(3) In the third clause of the example, the A in (λXA) must be a meta variable, since the intention of the clause is to allow capture of variables.

4.2. *Example: Defining Lambda Evaluation*

Consider a logic program to evaluate lambda expressions. The evaluator is to perform beta reduction, and is to prefer leftmost-outermost evaluation. We suggest that the following approach is natural.

$eval(A, C)$:- $step(A, B), eval(B, C)$.

$eval(A, A)$.

$step(((\lambda XA)B), \langle B/X \rangle A)$.

$step((\lambda XA), (\lambda XB))$:- *for-all* X $step(A, B)$.

$step((AB), (CB))$:- $step(A, C)$.

$step((AB), (AC))$:- $step(B, C)$.

Here are some indications of the semantics intended for this example.

(1) In the first clause of the *step* procedure, an interpreted substitution operator is used. That is, evaluation of $\langle B/X \rangle A$ by substitution of B for free occurrences of X in the instantiation of A is to be carried out at unification time. If A is uninstantiated, evaluation is delayed until A becomes instantiated. This substitution operation is a language primitive, not a user-defined quantifier such as λ, but it binds the object variable X. The object variables $Y \neq X$ which occur free in the instantiation of A are not captured. That is, they do not unify with X and so are not substituted. Changing the first clause to

$$step(((\lambda XA)B), A) :- X = B.$$

would not be appropriate, since then the occurs check would sometimes cause $X = B$ to fail. Intuitively, and formally in $((\lambda XA)B)$, occurrences of X which are free in some instantiation of A are within a different scope to the occurrences of X which are free in an instantiation of B.

(2) In the second clause of the *step* procedure, the subgoal $step(A, B)$ is quantified in order to specify that X is to be treated as a bound variable during the satisfaction of the subgoal. We have used universal-quantifier notation, which seems natural but is not essential. The essential point is that the quantifier *for-all* is interpreted. It directs the use of the appropriate unification algorithm, which does not instantiate X and which prohibits X from occurring in instantiations of object-level variables of $step(A, B)$. For brevity we call this *subterm* unification.

We do not claim that the above two examples illustrate all aspects of a logic-programming system to support quantified terms.

4.3. Conclusion

Although this paper stresses unification algorithms, it is important to provide also for transforming terms by arbitrary replacements, for example so as to support the efficient integration of functional and logic-programming techniques. Providing for replacements is not a big issue in the absence of structure sharing. Integrating our structure sharing with efficient replacement is however an interesting problem. It appears to be closely related to the problem of implementing J.-J. Levy's concept [1] of lambda-calculus family reduction.

Extension of the work of this paper to cover subterm unification is at an advanced stage. Extension to incorporate unification with meta variables is under way.

Orthogonal to the issues of meta variables, subterm unification and structure sharing are a range of issues which are familiar from the theory of free-variable unification, such as associativity and commutativity of appropriate function symbols. We have not yet addressed these issues, but there is no obvious reason why the free-variable algorithms should not extend to quantified terms. There are other issues too, in a similar vein but special to quantified terms, such as considering commutativity or idempotence of appropriate quantifiers. For example, *exists X exists Y A* is logically equivalent to *exists Y exists X A*, which can be recognized by an appropriate unification algorithm. Likewise *exists X exists X A* is logically equivalent to *exists X A*. Both these sorts of equivalence can be accounted for straightforwardly by refinements of our unification algorithm, but neither has been documented yet.

Added in proof: A preliminary report on work as indicated early in this section appears in [5].

Thanks to Richard Hagen for implementing the non-structure-sharing unification algorithm.

REFERENCES

1. Levy, J.-J., Reductions Correctes et Optimales dans le λ-Calcul, Thèse de doctorat d'état, Univ. Paris VII, 1978.
2. Spivey, M., *The York Portable PROLOG Interpreter*, University of York, 1984.
3. Staples, J. and Robinson, P. J., Structure Sharing for Quantified Terms, Univ. of Queensland Dept. of Computer Science Tech. Report No. 74, Nov. 1986.
4. Staples, J. and Robinson, P. J., Unification of Quantified Terms, Univ. of Queensland Dept. of Computer Science Tech. Report No. 70, June 1986.
5. Staples, J. and Robinson, P. J., Qu-Prolog—an Extended PROLOG for Symbolic Computation (to appear), Proc. Eleventh Australian Computer Science Conference, Brisbane, February 1988.