Order Independent Unification and Backtracking

Ian Clifford
John G. Cleary*

July 8, 1993

Abstract

Two algorithms for order independent unification are presented. Both assume that unifications are time stamped, giving a natural ordering on unifications from lowest timestamp to highest. The first algorithm permits lower timestamped unifications to be performed after higher timestamped unifications have already been done, without requiring previous unifications to be backtracked and redone. The binding state that results is indistinguishable from that produced by a lowest to highest execution. The second algorithm extends the first by allowing intermediate unifications to be undone, again avoiding work in undoing and redoing unifications with higher timestamps. These algorithms are designed for distributed, fully AND parallel Prolog execution, asynchronous events occurring on different CPUs can be merged [illegible] without too much overhead.

1 Introduction

The need for constructing and backtracking unifications in an arbitrary order has arisen within the context of AND parallel algorithms for Prolog. In an optimistic algorithm presented in [Clifford & Cleary 1992], all unifiers are associated with a virtual time. However, such unifiers, when passed between remote processors, may not arrive in timestamp order. An AND parallel system, [Cleary *et al* 1988] based on Time Warp [Jefferson & Sowizral 1985] dealt with such anachronisms by rolling back (backtracking) the bindings to the timestamp and then re unifying again. For anything other than the most deterministic programs, this backtracking and re execution lead to high overheads.

This paper shows how these overheads can be reduced by only incrementally modifying the binding environment when anachronistic messages arrive. The implication of these unification algorithms into the design of an AND-parallel Prolog system is described in [Clifford & Cleary 1993].

2 AND-Parallel Prolog

The main problem of AND parallel Prolog execution is in handling dependent goals - goals with one or more variables in common. In forward execution, two or more dependent goals may attempt to bind the same variable when unifying with their respective selected clauses. In backward execution, the backtracking behaviour of dependent goals must be controlled so that no potential solutions are lost.

Most AND parallel schemes solve the unification problem by imposing some fixed ordering on the execution of dependent goals. The *independent AND parallel* (IAP) systems [Cleary 1987, Lusk 1984, Kale 1985] use the depth first ordering of sequential Prolog - that is, dependent goals are executed sequentially (until the dependent variable is ground) according to their occurrence in the depth first search tree. Another class of systems, which includes the *concurrent logic programming* (CLP) languages [Clark & Gregory 1986, Shapiro 1987, Ueda 1987, Foster & Taylor 1990] and Somogyi's Prolog system [Somogyi *et al* 1988], uses mode annotations to determine the *producers* and *consumers* for each variable. Similar in spirit is Delta Prolog [Pereira *et al* 1986], in which

*Authors' address: Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, CANADA, T2N 1N4.
Email: clifford@cpsc.ucalgary.ca, cleary@cpsc.ucalgary.ca

producers communicate with consumers via *event goals*. Systems in this class exploit more parallelism than the IAP systems, because they permit *stream AND-parallel* execution [Conery & Kibler 1985], in which the consumers of a variable may execute as soon as the producer of that variable has even partially bound it. In both classes of system, however, full parallelism is not exploited. Both execute *conservatively*, and require dependent goals to wait for (full or partial) bindings for their shared variables.

The *optimistic* AND-parallel systems [Cleary *et al* 1988, Tebra 1989] offer true dependent AND-parallel execution. Each dependent goal *assumes* that it is the producer of each of its shared variables. In these systems, some goal ordering is still necessary—not an order of execution, but an order of binding precedence. If two goals attempt to bind a variable, the goal with higher precedence is accepted, and the goal with lower precedence is forced to *roll back*, rescinding its own binding and becoming a consumer of the other binding. Tebra's system bases its binding precedences on the depth-first goal ordering; Cleary *et al* base theirs on virtual times [Jefferson 1985] assigned at binding time.

Though some work may be deemed "wasted" due to rollback, the point is moot: goals rolled back in an optimistic system would have sat idle in a conservative one. Nevertheless, it is worthwhile to avoid rollback whenever possible. Olthof showed in [Olthof 1991] that an unoptimized system, though it performed well for fully determinate test programs, suffered when executing shallowly- and (especially) deeply-nondeterminate programs. Tebra's algorithm is optimized for forward execution, avoiding rollback whenever possible. However, his scheme is restricted to shared-memory architectures. Here we present an order-independent unification (OIU) algorithm that optimizes forward execution in the distributed-memory case, and is easily integrated with the basic algorithm of Cleary *et al*.

The backtracking problem in AND-parallel execution is more difficult. Some systems, specifically the CLP languages, avoid this problem by permitting no backtracking at all, thus explicitly departing from the full Prolog semantics. All of the other systems mentioned backtrack in the event of failure, and all must backtrack dependent goals sequentially (in reverse order of precedence), so that no potential solutions are missed.

All of the systems that backtrack also must be capable of rolling back. In the stream or dependent AND-parallel systems, a consumer (lower-precedence goal) may reject a binding from some producer (higher-precedence goal). The producer rolls back and withdraws its binding, causing each consumer of the binding to roll back. (The consumer that originally rejected the binding need not roll back, since it has already backtracked to the required state.) Even in the IAP systems (as well as in the others, of course), the producer may fail and withdraw its binding of its own accord, causing all consumers of the binding to roll back.

A rollback by a consuming process due to binding withdrawal is wasteful, because it forces such a process to undo a computation that will later be redone. Under the optimistic schemes, these rollbacks can be avoided by permitting bindings in a process to be backtracked in arbitrary order. To this end, we give a unification algorithm—the OIB algorithm—that permits order-independent unification and backtracking of bindings.

Both OIU and OIB algorithms assume that all bindings are timestamped. A binding's timestamp is used to determine its precedence relative to other bindings: a lower timestamp indicates a higher precedence. These timestamps are taken directly from the virtual times assigned each goal in the algorithm of [Cleary *et al* 1988, Olthof & Cleary 1992], in that the timestamp of a binding is given by the virtual time of the goal that produced it.

The rest of the paper is structured as follows. First, we give an algorithm for order-independent unification that for each variable retains only the lowest-timestamped binding for that variable, and discards later bindings. Next, we describe an algorithm for order-independent unification and backtracking that maintains information about multiple bindings for each shared variable. For each algorithm, we discuss the data structures required, the interface with the overlying Prolog system, and the expected overhead. We conclude with a look at the application of each algorithm within a Prolog system, and outline a distributed Prolog interpreter based on the second of these algorithms.

3 Order-Independent Unification

We present here an algorithm that avoids rollback as much as possible when unifying incoming bindings: the order-independent unification (OIU) algorithm. With a traditional unification algorithm, all bindings must be processed in timestamp order. Thus, when a binding with a certain timestamp arrives, the local execution must

be rolled back far enough to undo any bindings with higher timestamps. After the incoming binding is processed, the undone bindings can only be recreated through forward execution.

In contrast, the OIU algorithm attempts to insert each newly-processed binding into the stack according to its timestamp. If the variable being bound is unbound, the only effect is to add the new binding to the trail. If the variable is already bound, the situation becomes more complex, and two cases arise: either the bindings conflict, or they are compatible.

If the bindings conflict, then binding timestamps must be considered. If the existing binding has a lower timestamp, then the unification fails, and the incoming binding is rejected—but no rollback is necessary. If the incoming binding has a lower timestamp, a rollback is unavoidable, though it need only go back far enough to undo the existing binding, i.e. a *partial* rollback. The variable is then bound to the incoming binding, and the unification succeeds.

If the bindings are compatible, little need be done. When the incoming binding has the lower timestamp, the existing binding must be modified to reflect that earlier binding time. Otherwise, the existing binding stands. If the bindings are compound terms, each argument of the incoming binding is unified with its existing counterpart, and the success or failure of the entire unification rests on the success or failure of each sub-unification.

Since unification may require a partial rollback for success, some *rollback time* must be computed. Complete success results in a rollback time of $+\infty$, indicating that no rollback is necessary. Complete failure gives a rollback time equal to the timestamp of the incoming binding—that is, for the incoming binding to succeed, it must itself be rolled back, clearly a contradiction. A rollback time greater than the incoming binding time but less than $+\infty$ indicates the need for a partial rollback, after which the unification can succeed.

The only tricky part in calculating the rollback time comes in unifying compound terms. In this case, the rollback time of the entire unification is the minimum of the rollback times of each argument unification. The rollback time is computed incrementally, ending only when no more argument pairs remain to be unified (success) or when some argument pair fails to unify (failure).

Pseudocode for the OIU algorithm is given in Figure 1. The "@" notation used in this pseudocode deserves mention. The line

```
LD := RD @T
```

indicates that variable `LD` is being bound to term `RD` at virtual time `T`, and that the binding is recorded in the backtrack trail of the stack frame with time `T`.

In the sections that follow, the algorithm is described from several perspectives. First, we consider its interface to an overlying Prolog system. Next, we describe its dereferencing behaviour, and then we examine the unification algorithm in detail. At this point, we outline the data structures necessary to implement the algorithm. Finally, we discuss the time and space overheads that the algorithm incurs.

3.1 Prolog Interface

The unification algorithm is called from two places in a distributed-memory parallel Prolog interpreter. The first of these is in normal forward execution, in which case the algorithm behaves exactly like a standard Prolog unification algorithm. All bindings on the trail have lower timestamps (and thus higher precedence) than the bindings currently being created. Thus, the unification algorithm simply either succeeds or fails.

The second occasion for which unification is done is in receiving bindings from an interpreter on another machine. In this case, the incoming bindings may have higher precedence than some bindings already on the stack. Therefore, in addition to the the possibilities of success (no conflicts) and failure (conflict with a higher-precedence binding), there is also the possibility of conflict with a lower-precedence binding. If such a conflict occurs, it is necessary to roll back to the time of that lower-precedence binding.

The unification algorithm is given three parameters: `T`, the virtual time of the unification; `L`, the local binding term; and `R`, the remote (incoming) binding term. (In forward execution, `L` is the goal term and `R` is the clause term.) The algorithm returns a value indicating the success or failure of the unification.

The value returned by the algorithm is in fact the time to which the caller must roll back to remove any inconsistencies. If the value is infinity, no conflicts were found and the unification succeeded. If the value is less than or equal to the original unification time, then the unification failed—the new bindings would themselves have to be rolled back. Finally, if the value is somewhere between the two extremes, then the caller must roll back to the time given by that value to remove any conflicts.

```
unify(time T, term L, term R) returning rollback time RVAL

    (TL, LD) := deref(T, L)
    (TR, RD) := deref(T, R)

    if LD = RD
        return +inf
    else if LD is an unbound variable
        LD := RD @T
        return +inf
    else if RD is an unbound variable
        RD := LD @T
        return +inf
    else if LD is a bound variable
        if RD is a bound variable
            RVAL := unify(max(TL, TR), val(LD), val(RD))
            if RVAL > max(TL, TR) /* success */
                undo binding of LD
                LD := RD @T
            end if
            return RVAL
        else /* RD is a compound term */
            RVAL := unify(TL, val(LD), RD)
            if RVAL > TL
                undo binding of LD
                LD := RD @T
            end if
            return RVAL
        end if
    else if RD is a bound variable
        RVAL := unify(TL, LD, val(RD))
        if RVAL > TR
            undo binding of RD
            RD := LD @T
        end if
        return RVAL
    else if functors/arities match
        RVAL := +inf
        for each argument pair (LDi, RDi)
            RVAL := min(RVAL, unify(T, LDi, RDi))
            if RVAL = T
                return RVAL
            end if
        end for
        return RVAL
    else /* functors or arities don't match */
        return T
    end if
```

Figure 1: Pseudocode for order-independent unification

```
deref(time T, term VAR) returning pair (time T', term VAR')

    if VAR is bound
        Tb := binding time of VAR
        if T <= Tb
            return (Tb, VAR)
        else
            return deref(T, val(VAR))
        end if
    else /* VAR is unbound or a compound term */
        return (+inf, VAR)
    end if
```

Figure 2: Variable dereferencing

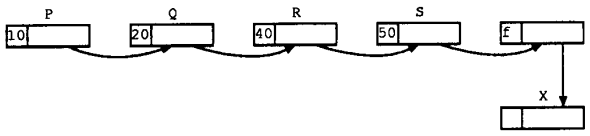

Figure 3: Binding structure before out-of-order unification

It should be noted that failure may occur partway through the unification of a group of incoming bindings. Because the unification algorithm alters existing bindings, care must be taken to ensure that the unification can be undone. Thus, any previous bindings must be retained until the whole unification has succeeded (with or without rollback). After this, the space consumed by the previous bindings may be reclaimed.

Because bindings may be altered, backtracking and rollback must also be performed carefully. A binding may have been trailed in some frame, and then had its binding time altered. Such a binding, after being untrailed from the removed frame (but *not* backtracked), must be appended to another trail: that of the stack frame whose virtual time is equal to the binding time. Only when this frame is removed will the binding actually be backtracked. In all other respects, backtracking and rollback are performed just as they are in systems using standard unification—e.g. if a binding with a certain timestamp is withdrawn, all bindings with later timestamps must be rolled back before the withdrawn binding is undone.

3.2 Dereferencing

During the course of execution, a variable may become bound to another unbound variable, and then to another, and another .... This gives rise to a reference chain that must be dereferenced when the variable is bound again to a new value (either a compound term, a constant, or another variable). Thus, the first step in the OIU algorithm, as in most unification algorithms, is to dereference both binding terms. (Remote or clause term dereferencing is unnecessary at the top level, but subterms may well require dereferencing.) In standard algorithms, a term is completely dereferenced—to an unbound variable or an integer, atom, or compound term. (For the balance of the paper, constants—integers and atoms—are treated as compound terms with arity zero.)

Under the order-independent algorithm, a term might not be completely dereferenced. Instead, it is dereferenced only until the binding time of the current alias is greater than the current unification time. This ensures that when a reference chain is formed or extended, the bindings of the variables in the chain are strictly increasing in timestamp from the beginning of the chain to the end. In this way, the chain appears to have been built up in timestamp order, and may be backtracked in the "opposite" order without destroying essential binding information.

Pseudocode for the dereferencing algorithm is shown in Figure 2. A (presumed) variable `VAR` is dereferenced with respect to some virtual time `T`. If `VAR` is an unbound variable or a compound term, it cannot be dereferenced, and is simply returned along with a "binding time" of $+\infty$. Otherwise, `VAR` is bound to some other term, so its binding time, `Tb`, must be examined. If `T` exceeds `Tb`, dereferencing continues with `VAR`'s binding value. Finally, if `Tb` exceeds `T`, dereferencing stops, and `VAR` is returned along with its binding time.

To see how this works, consider the situation depicted in Figure 3. Variable `P` is bound (via a long reference

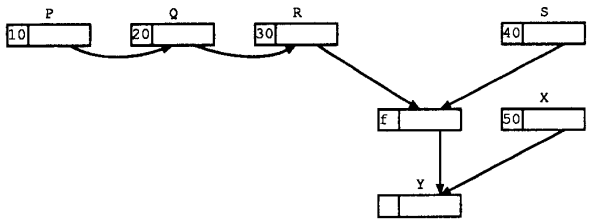

Figure 4: Binding structure after out-of-order unification

chain) to the term `f(X)`. The binding `P = f(Y)` arrives at virtual time 30. When `P` is dereferenced, the dereferencing algorithm returns the bound variable `R` and the time 40. The unification algorithm then calls itself recursively to unify the remainder of the reference chain (starting at `S`) with `f(Y)` at time 40.

`S` is itself dereferenced, but not very far: the dereferencing algorithm returns `S` immediately, with binding time 50. Again, the remainder of the reference chain, `f(X)`, is unified with `f(Y)`, this time at time 50. This unification results in the binding of `X` to `Y` at time 50. Unwinding the recursion, `S` has its binding at time 50 to `f(X)` removed, and is rebound to `f(Y)` at time 40. `R` then receives similar treatment, losing its binding to `S` at time 40, but being bound to `f(Y)` at time 30. The resulting binding structure is shown in Figure 4; it is identical to the structure that would have resulted from undoing the bindings at times 40 and 50, and then performing unifications at times 30, 40, and 50.

3.3 Unification Algorithm in Depth

After dereferencing, unification begins in earnest. Each term may have been dereferenced to one of three types: an unbound variable, a bound variable, or a compound term. This gives rise to nine cases; the pseudocode of Figure 1 handles each of these cases.

The simplest case occurs when both dereferenced variables (`LD` and `RD`) are unbound. Either they are identical, or one must be bound to the other (by convention, `LD` is bound to `RD`) at time `T`. In either case, the unification succeeds, returning a rollback time of $+\infty$. When one term is unbound and the other is a compound term (two cases), the unbound variable is simply bound to the compound term with binding time `T`. Similarly, when one term is unbound and the other bound (also two cases), the unbound variable is bound to the bound variable with binding time `T`.

The remaining cases deal with unifying various combinations of bound variables and compound terms. The simplest of these comes in unifying two compound terms. No new bindings are made at the top level. If the terms differ in functor or arity, the unification fails, returning rollback time `T`. Otherwise, the rollback time is computed as the minimum of the return values from each pairwise argument sub-unification, and returned. (Atoms and integers are deemed to have an arity of zero; thus, no sub-unifications are done and the computed "minimum" is $+\infty$.)

The final three cases—two of which unify a bound variable with a compound term, and one of which unifies two bound variables—require one or more bindings to be altered. When a bound variable `B` is unified with a compound term `C` at a unification time `T` (as in the example of Figures 3 and 4), a sub-unification must first be done. `C` is unified with the remainder of `B`'s reference chain, with a unification time equal to `B`'s binding time. If this sub-unification succeeds, `B`'s previous binding is undone, and replaced with a binding to `C` at time `T`. (The previous binding, though compatible, had a greater virtual time, and thus would not have been created had the unifications occurred in virtual time order.) If the sub-unification fails, this modification is not performed. In both cases, the rollback time received from the sub-unification is used as a return value.

Finally, when two bound variables are unified (assuming they are not identical), the remainders of their respective reference chains are unified first. If this succeeds, then one of the bound variables (by convention, `LD`) must have its old binding undone and be bound to the other (`RD`) at time `T`. If the sub-unification fails, no rebinding is done. Again, the return value is given by the rollback time returned by the sub-unification.

It should be noted here that, like most other unification algorithms, the OIU algorithm performs no occurs

check. This also applies to the OIB algorithm, described in Section 4.

3.4 Data Structures

With respect to the data structures employed, the OIU algorithm differs from standard unification algorithms in three ways. First, a timestamp must be associated with each variable binding, so that binding conflicts can be resolved. Second, frames may be inserted at arbitrary points in the execution history (normally, the stack). Third, the backtrack trail associated with each stack frame must be extensible.

If the timestamp information is simple enough (say, representable by a single integer), it may be kept directly as part of each binding. This provides fast access to binding times during unification. If a timestamp consumes several machine words, it may be more reasonable (in terms of space overhead) to have each binding contain a pointer to the frame in which it was bound, and to store the timestamps only in the stack frames. Accessing a binding time on unification then requires a pointer dereference.

Because of the requirement for arbitrary insertion, a stack is no longer practical for maintaining the execution history. Inserting a frame at some time `t` would require that all frames with timestamps greater than `t` be popped, the new frame pushed, and the popped frames pushed back on the stack. More practical would be a linked list of frames, allowing arbitrary insertions (and deletions). Since most operations on this frame list remain "pushes" and "pops" from the end of the list, and in the interest of terminological nonproliferation, we shall continue to refer to it as a "stack," and trust that no confusion will result.

Finally, when a frame with some timestamp `T` is rolled back or backtracked, any variable binding in its trail with a binding time of less than `T` must not be undone. Rather, it must be trailed again, in the frame corresponding to its binding time. That is, if a variable is originally bound at virtual time `T` and later bound at time `T - k`, then when the frame with timestamp `T` is removed due to rollback or backtracking, the variable must be trailed in the frame with timestamp `T - k`.

3.5 Overhead

The OIU algorithm incurs very little overhead over that of the standard Prolog unification algorithm. With respect to space consumption, the only additional costs are in a timestamp for each binding, and in pointers to maintain the frame list. In forward execution, the only time cost is in setting the binding time of each binding produced. In processing incoming bindings, the comparison of binding times adds a small constant overhead for each bound variable unified with a compound term or another bound variable.

Otherwise, the only additional work is in adjusting bindings and binding times. Such adjustments occur when an incoming binding takes precedence over an existing binding, whether that binding is compatible or not. Unifying a variable with some binding may require several adjustments if that variable is already bound to a compound term or if it is part of a variable reference chain. Finally, unifications involving unbound variables incur neither the timestamp comparison cost nor the binding adjustment cost.

4 Order-Independent Backtracking

Although the OIU algorithm optimizes the addition of bindings to the trail in arbitrary order, it does nothing to mitigate the effects of *removing* such bindings. If a binding at some time `T` is withdrawn, all work done at later virtual times must be rolled back, and later redone. As an alternative, we present the OIB algorithm, which optimizes rollbacks due to binding withdrawal in addition to those due to binding arrival.

This further optimization comes at a price, however. In the case of the OIU algorithm, bindings can be added in any order, and the resulting state will be identical to the state that would be reached by unifying these bindings in virtual time order. Thus, only this "standard" state need be maintained. Because it allows bindings to be removed arbitrarily, the OIB algorithm must be able to go from a given state to a previous state. In fact, all *possible* previous states must be accessible, not just actual previous states.

To accomplish this, we keep redundant binding information. Each bound variable has associated with it an *active* binding and a list of *inactive* bindings. The active binding for a variable is the lowest-timestamped binding for that variable, while the inactive bindings are all those with greater timestamps. The set of all active bindings

```
unify(time T, term L, term R) returning (rollback time RVAL, set DEP)

    if L = R                              /* terms are identical, all done */
        return (+inf, {})
    else if L is a bound variable         /* L is bound, try to deref */
        LB := binding value of L
        TL := binding time of L
        if T >= TL                        /* new binding is later, so deref */
            return deref(T, L, LB, R)
        else if R is a bound variable     /* R is bound, try to deref it */
            RB := binding value of R
            TR := binding time of R
            if T >= TR                    /* new binding is later, so deref */
                return deref(T, R, L, RB)
            else if TL >= TR              /* L bound later than R, so rebind L */
                return rebind(T, L, R, TL, LB)
            else                          /* R bound later than L, so rebind R */
                return rebind(T, R, L, TR, RB)
            end if
        else                              /* R is a compound term, so rebind L */
            return rebind(T, L, R, TL, LB)
        end if
    else if R is a bound variable         /* R is bound, try to deref */
        RB := binding value of R
        TR := binding time of R
        if T >= TR                        /* new binding is later, so deref */
            return deref(T, R, L, RB)
        else                              /* new binding is earlier, so rebind */
            return rebind(T, R, L, TR, RB)
        end if
    else if L is an unbound variable      /* terms are distinct, L is unbound */
        L := R @T                         /* => bind L */
        return (+inf, {(T,L)})
    else if R is an unbound variable      /* terms are distinct, R is unbound */
        R := L @T                         /* => bind R */
        return (+inf, {(T,R)})
    else if functors/arities match        /* both L and R are compound terms */
        RVAL := +inf                      /* init return value */
        for each argument pair (Li, Ri)   /* unify arguments pairwise */
            (RV, D) := unify(T, Li, Ri)
            if RV = T                     /* on failure, quit right away */
                return (RV, {})
            end if
            RVAL := min(RVAL, RV)         /* get lowest rollback time */
            DEP := union(DEP, D)          /* keep track of dependent bindings */
        end for
        return (RVAL, DEP)
    else                                  /* functors or arities don't match */
        return (T, {})                    /* return failure */
    end if
```

Figure 5: Main unification routine for order-independent backtracking

corresponds exactly to the state computed by the OIU algorithm; the inactive bindings constitute redundant information.

Bindings must be convertible from active to inactive and back. Active bindings are explicitly recorded as substitutions, like normal Prolog bindings. Inactive bindings are recorded as pending unifications. When a new binding arrives whose timestamp is lower than any other binding for the same variable, it must become the new active binding, and the existing active binding must be converted to an inactive binding. When an active binding is removed, the lowest-timestamped inactive binding (if it exists) for the same variable must become the new active binding. These actions replace respectively the binding-time adjustment and rollback done by the OIU algorithm in the same situations.

Pseudocode for the OIB algorithm is given in Figures 5 through 8. The algorithm consists of four main routines: `unify` (Figure 5), `deref` (Figure 6), `rebind` (Figure 7), and `undo` (Figure 8). These routines function as follows:

`unify(T,L,R)` unifies terms `L` and `R` at virtual time `T`. It decides what to do with each term depending on its type and (if bound) its timestamp.

```
deref(time T, term L, term R, term B) returning (time RV, set D)

    (RV, D) := unify(T, R, B)           /* unify with the deferenced term */
    if RV > T                           /* successful unification */
        DEP(L) := union(DEP(L), D)      /* add in new dependent bindings */
        insert(INACT(L), (T, L, R))     /* new binding is inactive at top level */
        return (RV, {(T,L)})            /* return success and inactive binding */
    else
        undo(D)                         /* unification failed, undo */
        return (RV, {})                 /* return failure */
    end if
```

Figure 6: Dereferencing in OIB unification

```
rebind(time T, term L, term R, time TB, term B) returning (time RV, set D)

    unbind(L)                           /* remove old binding first */
    L := R @T                           /* install new binding */
    (RV, D) := unify(TB, B, R)          /* unify state further with new binding */
    DEP(L) := union(DEP(L), D)          /* add in new dependent bindings */
    insert(INACT(L), (T, L, B))         /* old binding becomes inactive */
    return (RV, {(T,L)})                /* return success and new binding */
```

Figure 7: Variable rebinding in OIB unification

```
undo(dependent set D)

    for all (T,V) in D
        undo(DEP(V))                    /* undo all subsidiary bindings */
        unbind(T,V)                     /* remove a single binding */
    end for
```

Figure 8: Removal of dependent bindings in OIB unification

`deref(T,L,R,B)` dereferences variable `L` by one step to term `B`. It then calls `unify` to unify term `R` with `B`.

`rebind(T,L,R,TB,B)` removes a binding `B` for variable `L` at virtual time `TB`, and replaces it with a new binding, `R`, with a lower timestamp, `T`. The old binding is then unified with the new via a call to `unify`, to ensure their compatibility.

`undo(D)` backtracks a set of bindings `D`, all of which depend on the existence of some other binding (and which must therefore be removed on the withdrawal of that binding).

4.1 Prolog Interface

The interface to a Prolog caller is very similar to that for the previous algorithm. Like the OIU algorithm, the OIB algorithm is also called in normal forward execution, and in processing incoming bindings. Both receive as parameters a unification time and two terms to unify. Both compute and return to the caller a rollback time, `RVAL`, indicating the success or failure of the unification.

Unlike the previous algorithm, however, the OIB algorithm is also called when an active binding is withdrawn. When this occurs, the lowest-timestamped inactive binding for the same variable becomes the new active binding—not via rollback, as in the OIU algorithm, but rather by undoing the withdrawn binding and then unifying the new active binding with the current state.

Also unlike the OIU algorithm, the OIB algorithm computes and returns a set of dependent bindings `DEP`. These are bindings which were made as a direct consequence of performing the top-level unification. When the top-level unification is undone, any bindings in the dependency set must be undone as well. (In the OIU algorithm, such dependent sets were unnecessary because any bindings created as a consequence of performing some unification would automatically be rolled back if that unification were undone.)

As in the OIU algorithm, failure may occur partway through a unification. In this case, however, no special care is needed to clean up any partial unifications, since the partially-computed set of dependent bindings, `DEP`, is also returned on failure. Thus, failure can be treated as binding withdrawal, and any bindings due to the failed unification can be accessed through the returned dependent set and undone.

4.2 Dereferencing

As with most unification algorithms, the OIB algorithm requires some way of dereferencing variable reference chains, so that the last variable in a chain is the one bound, and accesses to any variable in the chain will return the proper binding. Most unification algorithms perform this dereferencing step explicitly, before performing the actual unification.

Under the OIB scheme, dereferencing is integrated with the unification process itself. This is because the OIB algorithm must record an inactive binding for each dereferencing step, rather than just searching down the chain. A simple example illustrates this behaviour.

Suppose that at virtual time 30 we perform the binding `P = Q`, at time 40 the binding `X = Y`, and at time 50 the binding `P = 7`. Under the OIU and standard schemes, the last unification would deference `P` to `Q` and cause `Q` to be bound to `7`. The information that the last binding was originally made to `P` can be discarded safely, since the last binding will always be removed before the first.

Under the OIB algorithm, this is no longer the case. The idea is that the binding `P = Q` can be removed without having to undo all later bindings, only to redo all those bindings to restore the proper state. Instead, only the withdrawn binding—and each of the bindings that depend on it—is undone, and the first available inactive binding is recorded in its stead. Thus `P = 7` must be recorded as an inactive binding for `P` when it is first encountered, as well as causing `Q = 7` to become the active binding for `Q`. If `P = Q` is later removed, the binding for `Q` disappears, and `P = 7` becomes the active binding for `P`, without rolling back and disturbing the binding `X = Y`.

In other respects, dereferencing behaviour is similar to that of the OIU algorithm. In particular, a reference chain may not be dereferenced completely: dereferencing stops when a binding is encountered in the chain whose binding time is greater than the unification time. Thus, a "fully-dereferenced" variable may still be bound.

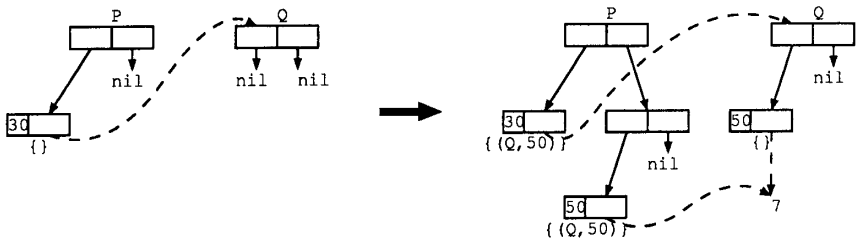

Figure 9: Calculation of dependent sets

4.3 Dependent Sets

Associated with each binding for a variable is a dependent set, which is initially empty. As a result of unifying each previous binding for the variable with the new binding, this dependent set is augmented to include subsidiary bindings that are directly due to the new binding—thus, if the new binding is removed, so will all the subsidiary bindings. If the new binding is to a nonground term, such as another variable or an incomplete compound term, the dependent set will also be augmented each time a dereferenced or subterm variable is bound due to a unification that references the nonground binding.

Consider the following execution fragment, illustrated in Figure 9 and based on the example of the previous section. Solid arrows in the diagrams denote pointers in the binding structure for each variable, while dashed arrows denote bindings. The left-hand diagram illustrates the initial state: at virtual time 30, P is bound to Q. Then, at time 50, P is bound to 7. As a result, Q is also bound to 7 at time 50, resulting in the state shown in the right-hand diagram.

This binding of Q is dependent on both bindings of P, and if either were withdrawn, it must be withdrawn itself. That is, if either of P = Q or P = 7 is undone, the binding Q = 7 must be undone. Thus, the binding for Q is put in the dependent sets of P's bindings at times 30 and 50. The dependent sets are used in backtracking, so that when a binding is undone, any bindings that depend on it—exactly the bindings in the dependent set—are also undone.

4.4 Unification Algorithm in Depth

The main unification routine is structured much like that for the OIU algorithm. The terms L and R may each be either an unbound variable, a bound variable, or a compound term. The algorithm thus decomposes into several cases, each of which returns two values: a rollback time as in the OIU algorithm, and a set of dependent bindings.

The simplest case occurs when L and R are identical. The unification succeeds immediately, with a rollback time of $+\infty$ and an empty dependent set. Also simple are the cases in which either or both of L and R are unbound. If L is unbound, it is bound to R at time T returning a rollback time of $+\infty$ and a dependent set consisting of the binding itself. Otherwise, if R is unbound, it is treated in a similar manner. Because dereferencing is integrated into the algorithm, both L and R must be checked first to see whether either is bound. Otherwise an unbound variable could be bound to some undereferenced bound variable, creating a reference chain that violates the increasing-timestamp condition described in Section 3.2.

The remaining cases are more complex. If either or both of L and R are bound variables, some dereferencing may be necessary. As with the OIU algorithm, dereferencing is required if a variable's binding time (TL or TR in the pseudocode) is less than the unification time T. If L is a bound variable, it is dereferenced as much as possible; then the same is done for R.

When a variable has been dereferenced as much as possible and the term finally reached is still a bound variable (*i.e.* the binding time of that final variable is greater than the unification time), then the final variable must be *rebound* to the new value. When this is done, the old value must be unified with the new value. If this unification succeeds, the old value is retained as an inactive binding (otherwise, it is simply discarded). When both dereferenced terms are still bound, one must be chosen to be rebound. This is done on the basis of binding age: the term with the greater binding time of the two is the one that is rebound.

Finally, when both terms are compound terms, functors and arities are compared. On success, corresponding arguments from each term are unified, with return values and dependent bindings being accumulated in the process.

As well, several primitives are used in the pseudocode. `Union(S1,S2)` gives the union of sets `S1` and `S2`; `insert(L,B)` inserts an inactive binding `B` into a list `L` of such bindings (sorted in timestamp order); `unbind(T,V)` backtracks the binding for `V` at time `T`. `DEP(L)` refers to the set of bindings dependent on `L`, while `INACT(L)` refers to the list of inactive bindings for `L`.

4.5 Data Structures

Like the order-independent algorithm for unification alone, the OIB algorithm needs a timestamp associated with each binding in order to determine the success or failure of a unification and to determine a rollback time if binding conflicts are detected. The "stack" must again be a list, allowing the addition and deletion of internal frames.

As well, each binding must maintain a list of other bindings that depend on it, so that these dependent bindings can be backtracked when the original binding is backtracked. A binding may depend on more than one other binding, particularly when variables are bound to other variables (aliasing) or to incomplete compound terms. In the pseudocode, the set of dependent bindings dependent on the binding of `R` is known as `DEP(R)`.

To maintain information about inactive bindings, each variable has associated with it a timestamp-ordered list, referred to in the pseudocode as `INACT(L)` for some variable `L`. Each element of the list constitutes an inactive binding.

4.6 Overhead

Overhead for the OIB algorithm is somewhat higher than for other unification algorithms, including the OIU algorithm. This is mostly due to the cost of installing and removing inactive bindings and dependent sets. The remaining overheads are identical to those faced by the OIU algorithm. The execution "stack" is actually a list, so that the arbitrary insertion and removal of stack frames is possible. To determine binding conflicts, each binding must include a timestamp; this timestamp must be examined each time the binding is accessed.

In any case, the cost of doing an OIB unification will be linear in the cost of a standard unification. Also, the cost of doing a binding removal can be arbitrarily lower than the corresponding backtrack and re-unification of standard Prolog. In the worst case, the OIB cost will be linear in the standard cost (presumably with a somewhat lower constant).

In cases where all bindings occur in timestamp order, OIB unification will behave almost identically to standard unification, although dependent sets will be created and variables will have multiple bindings recorded. Even this cost can be ameliorated in the usual case. Consider a variable bound as a result of a remote unification from another processor. Any local attempt to match this variable with a clause head will cause an inactive binding to be recorded. However, this local binding can only be removed by backtracking, so later-timestamped bindings—both local and remote—need not be recorded. This optimization means that the most common case will be a variable bound with a single (active) binding marked as being local. Next most likely is a variable with an active binding caused by a remote unification plus one local (inactive) binding. More complex binding situations require relatively unlikely timing coincidences between remote processors.

5 Conclusion

Both the OIU algorithm and the OIB algorithm may be used as part of a dependent AND-parallel Prolog system. The Prolog system must provide the following facilities:

- a timestamp for each requested unification
- a rollback facility to handle binding conflicts
- direct access by the unification algorithm to the stack and backtrack trail

The first of these is based on the precedence of the goal being unified. The second is quite similar to backtracking, except that rollback is done to a specific virtual time, rather than back to the next untried clause. The last facility may be provided by adding suitable parameters to the unification call.

The OIU algorithm is designed for deterministic execution, in which little backtracking or rollback occurs. In this situation, effort is concentrated on optimizing forward execution as much as possible without introducing substantial overhead. A good Prolog implementation based on the OIU algorithm should be competitive with any of the concurrent logic programming languages.

In contrast, the OIB algorithm is designed more for problems in which backtracking and rollback are likely, but also in which AND-parallelism can be exploited. In this case, backward execution also merits optimization, even at the cost of some additional overhead in forward execution. With care, this overhead can be minimized, as outlined in [Olthof & Cleary 1992].

In that paper, we describe the design of a Prolog system based on the OIB unification algorithm. This system uses Time-Warp style [Jefferson 1985] timestamping, message passing, and rollback to achieve full dependent AND-parallel execution. An interpreter-based distributed implementation of this system is planned. After this, test results should be quickly forthcoming.

We also believe that the OIB algorithm could be of use when doing intelligent backtracking. The timestamp information maintained by both the OIU and OIB algorithms provides a good basis for intelligent backtracking [Mannila & Ukkonen 1986]. As well, OIB provides the ability to remove early stack frames without disturbing intervening partial solutions.

6 Acknowledgements

This work was supported by the Natural Science and Engineering Research Council of Canada.

References

[Clark & Gregory 1986] K.L. Clark and S. Gregory. PARLOG: parallel programming in logic, *ACM TOPLAS*, 8(1):1–49, 1986.

[Cleary *et al* 1988] J.G. Cleary, B.W. Unger, and X. Li. A distributed AND-parallel backtracking algorithm using Virtual Time, In *Proceedings of the Distributed SImulation Conference*, pages 177–182, San Diego, February 1988.

[Conery 1987] John S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.

[Conery & Kibler 1985] John S. Conery and Dennis F. Kibler. AND parallelism and nondeterminism in logic programs, *New Generation Computing*, 3(1):43–70, 1985.

[DeGroot 1984] Doug DeGroot. Restricted AND-parallelism, In *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 471–478, Tokyo, November 1984.

[Foster & Taylor 1990] Ian Foster and Steve Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.

[Jefferson 1985] David R. Jefferson. Virtual Time, *ACM TOPLAS*, 7(3):404–425, July 1985.

[Jefferson & Sowizral 1985] David R. Jefferson and Henry A. Sowizral. Fast concurrent simulation using the Time Warp mechanism, In *Proccedings of the SCS Distributed Simulation Conference*, San Diego, CA, January 1985.

[Kalé 1985] L.V. Kalé. *Parallel Architectures for Problem Solving*. PhD thesis, Department of Computer Science, SUNY Stony Brook, 1985.

[Mannila & Ukkonen 1986] Heikki Mannila and Esko Ukkonen. Timestamped term reprsentation for implementing Prolog, In *Proceedings of the 1986 Symposium on Logic Programming*, pages 159–165, Salt Lake City, Utah, 1986.

[Olthof 1991] Ian Olthof. *An Optimistic AND-Parallel Prolog Implementation.* Master's thesis, Department of Computer Science, University of Calgary, 1991.

[Olthof & Cleary 1992] Ian Olthof and John G. Cleary. The Design of an Optimistic AND-Parallel Prolog. Research Report 92/474/12, Department of Computer Science, University of Calgary, May 1992.

[Pereira *et al* 1986] L.M. Pereira, L. Monteiro, J. Cunha, and J.N. Aparício. Delta Prolog: a distributed backtracking extension with events, In *Proceedings of the Third International Conference on Logic Programming*, pages 69–83, 1986. published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Shapiro 1987] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter, In *Concurrent Prolog—Collected Papers*, chapter 2, pages 27–83. MIT Press, 1987.

[Somogyi *et al* 1988] Z. Somogyi, K. Ramamohanarao, and J. Vaghani. A stream AND-parallel execution algorithm with backtracking, In *Fifth International Conference and Symposium on Logic Programming*, pages 386–403, Seattle, August 1988.

[Tebra 1989] Hans Tebra. *Optimistic AND-Parallelism in Prolog.* PhD thesis, Vrije Universiteit, Amsterdam, 1989.

[Ueda 1987] K. Ueda. Guarded Horn clauses, In *Concurrent Prolog—Collected Papers*, chapter 4, pages 140–156. MIT Press, 1987.