The Pennsylvania State University
University Scholars Program
Department of Computer Science

Transformation of Prolog Programs to Perform Update in Place: A Prototype Code Synthesizer

Benjamin R. Liblit

April 30, 1993

A thesis submitted in partial fulfillment
of the requirements of the degree of
Bachelor of Science
with honors in Computer Science
and from the University

William H. Winsborough
Thesis Supervisor
Honors Chairman

Barry Pangrle
Honors Advisor

Mary Jane Irwin
Head of Computer Science

Abstract

Single assignment languages, such as Prolog or Standard ML, endure considerable inefficiency to support their declarative semantics. Lacking destructive assignment, programs written in these languages must frequently duplicate large portions of structures that conventional programs would simply modify in place. This paper presents techniques for automatically transforming Prolog programs to perform in-place updates using destructive assignment, and reports and evaluates experience prototyping the code synthesis portion of the transformation. Where applicable, such a transformation produces code that updates structures two to more than three times faster than naïve implementations and uses vastly less memory.

Contents

List of Figures

List of Tables

1 Introduction

Logic and functional programming languages have, until recent years, been restricted to the realms of academia and formal theory. The expressive simplicity and high level of abstraction that make these languages attractive also tend to make them inefficient. It is difficult for Prolog, Standard ML, and related languages to compete with more conventional languages such as C or Fortran. One major cause for this inefficiency is the extra work required to support single assignment semantics. When an incremental change is to be made to some large structure, a multiple assignment language would perform the updates in place, modifying only those fields that need to be changed. A single assignment language, on the other hand, would be forced to copy much of the structure, replacing old values with new ones as it went. The overhead associated with allocating new memory, duplicating data, and garbage collecting old, discarded copies hurts performance considerably.

If one can detect that the original copy of a structure is never reused past the point of modification, it should be possible to update it in place, destructively modifying it to form the new structure we wish to generate. We refer to such structures as *dead*, and it is the memory cells occupied by these soon-to-be discarded structures that we intend to reuse. More specifically, if static analysis can detect that certain arguments to a predicate or function are always dead, the memory cells occupied by these arguments may be "recycled" to construct the predicate or function's output, or to construct arguments to body goals or subcalls. The problem of detecting deadness relates directly to that of tracking liveness, and several researchers have developed static analysis techniques that can be applied toward this end [1, 2, 3, 4]. Certain languages, including Prolog, do not explicitly identify output

arguments. In such cases, we define the "output" of a predicate to be those arguments that are passed free variables wherever the predicate is called. Again, this may be determined via static analysis, using methods described in [5] and [6].

Ideally, one would like to allow the programmer to continue to use logic and functional languages without giving up the elegance and ease of expression that single assignment affords. The compiler would synthesize equivalent code that performs multiple, destructive assignment. Augmented by primitives for performing simple, in-place assignments, the transformed source should be far more efficient, using much less memory and potentially executing at speeds competitive with those of conventional languages.

Earlier attempts at applying multiple assignment efficiency to single assignment languages have demonstrated that this class of optimizations can lead to significant improvements [7, 2, 8]. However, the multiple assignment code produced using these prior approaches may not always operate as efficiently as one would like. These techniques are essentially equivalent for the purposes of this paper, and we will therefore focus our review on Compile Time Garbage Collection (CTGC), the approach described in [8].

An alternative technique, the one with which this paper is centrally concerned, is the Update in Place (UIP) transformation. Like CTGC, the UIP transformation produces code that is able to reuse dead memory cells. However, UIP transformed programs avoid the difficulties that hinder CTGC code's performance, yielding still faster code. The complete UIP transformation is a three stage process. The first stage performs dataflow analyses of the source program. During this phase, deadness detection is performed, and input and output arguments are identified. The second stage performs reuse analyses. Dead input cells are matched up with the output structures they will be used to create, forming a mapping

that describes specifically how and where memory cells will be reused. The problem of how to produce good reuse mappings is an important open question; Debray [6] has proven that a related reuse problem is NP complete but that efficient heuristics can usually find good mappings. It is yet unclear whether the slightly different reuse problem faced by UIP shares these traits. See [9] for further discussion of the relationship between Debray's work and UIP.

The third and final stage of the UIP transformation is code synthesis. Based upon the dataflow patterns deduced in the first stage, and using the reuse mappings selected in the second stage, the code synthesizer transforms single assignment programs into equivalent programs that destructively reuse memory cells to create new structures. A prototype code synthesizer for programs written in Prolog has been developed based upon formal specifications given in [10]. This paper will primarily address the implementation issues involved in creating such a synthesizer. Timing and memory use data are provided that directly compare the performance of naïve code, CTGC code, and several variations of UIP code on a small collection of benchmarks.

Section 2 reviews CTGC, focusing on those contexts in which it produces code that does more work than is necessary. Section 2 also develops an intuitive basis for the type of transformation we wish to apply here, contrasting it with CTGC. Section 3 discusses several implementation issues that must be resolved to produce transformed code that is not only correct, but highly efficient as well. Section 4 presents empirical studies of the performance of a transformed collection of benchmarks, and evaluates the impact of these issues. Lastly, Section 5 summarizes the author's findings and discusses broader issues regarding synthesizer and its incorporation into a working UIP transformation system.

While the transformation we present is fairly general, we have chosen Prolog as a working language in which to implement and describe this research. For a discussion of cross language issues, see [9].

2 Background

This section introduces previously presented techniques for rewriting Prolog predicates and clauses to reuse the memory cells of dead input structures. The first subsection reviews Compile Time Garbage Collection and illustrates that it may produce code that does extra work, hampering its efficiency. The second subsection introduces the present proposal, Update in Place (UIP). We will see that UIP avoids the pitfalls of earlier attempts, resulting in faster, more efficient transformed code.

2.1 Compile Time Garbage Collection

The prior proposal for transforming Prolog programs to reuse dead memory cells is Compile Time Garbage Collection [8]. CTGC destructively modifies cells to be reused so that they look like free variables newly allocated from the heap. Memory cells that would ordinarily be discarded are instead made available for immediate reuse. This reduces the garbage collector's work load, leading to faster, more memory efficient code. Furthermore, any fields that have the same value when the reused cell is reinstantiated need not be reinitialized. Closer examination reveals that CTGC transformed code may perform needless work nonetheless. First, consider a naïve implementation of `append`/3, which concatenates its first two arguments into the third:

```
append( [ Head | Tail ], List, [ Head | NewTail ] ) :-
    append( Tail, List, NewTail ).

append( [], List, List ).
```

Assume that static analysis has determined that the first and second arguments are always dead after calls to `append`/3, and that the third argument is always a free variable in all calls. Both Compile Time Garbage Collection and Update in Place attempt to reuse the list cells from the first and second arguments to construct the third. CTGC's approach is to make potentially reassignable fields of reused cells into free variables. Essentially, fields that we might wish to modify are *un*instantiated, so that they may be *re*instantiated with new values.

The following code realizes the CTGC transformation of `append`/3. It is worth noting that CTGC was originally proposed as an optimization of Warren Abstract Machine code. For the purpose of contrasting it with UIP, though, CTGC is more conveniently presented as a source level transformation.

```
append( Front, Rear, Front ) :-
    Front = [ _ | Tail ],
    makefree( 2, Front, NewTail ),
    append( Tail, Rear, NewTail ).

append( [], Rear, Rear ).
```

CTGC uses a new primitive, `makefree`/3, to rewrite fields as free variables. When this primitive is called as `makefree( +ArgNum, +Term, -FreeVar )`, it destructively modifies the `ArgNum`'th field of `Term` so that it resembles an uninstantiated free variable. A reference to this newly freed variable is placed in `FreeVar`.

In the CTGC code for `append`/3, the second field of `Front` is made into a free variable referenced by `NewTail`. This works well when the next element in the list is nil: the second

clause of the transformed predicate instantiates `NewTail` to the list `Rear`, and the output argument has been completed. However, consider what happens when the list does not yet terminate, and the first transformed clause executes again. Head unification unifies the first and third arguments, which were `Tail` and `NewTail` in the preceding level of recursion. Thus, `NewTail` is reinitialized to reference the list referenced by `Tail`, which is exactly what it had been referencing before the call to `makefree`/3. The time spent destructively unbinding and then rebinding `Front`'s second field has been wasted, since for most of `append`/3's execution, that second field is immediately reinstantiated to the same value it already held. Clearly, transformed code would perform faster if it did not spend time preparing to change fields that ultimately end up staying the same.

2.2 Update in Place

2.2.1 Last References

Compile Time Garbage Collection does extra work because it modifies reusable cells too early. The Update in Place transformation avoids this pitfall by performing its destructive assignments lazily, only where and when necessary. This is accomplished by replacing the output argument in each clause of the original predicate with a reference to the most recent cell that *may* need to be destructively modified. However, no assignment is made until it is genuinely required. By convention, we refer to this extra argument as a *last reference*, and refer to it in transformed code using variable names such as `Last`. Intuitively, it is the last cell that has been recycled into the output structure on a given level of recursion.

To better understand the need for a `Last` argument, consider how a program written in a conventional language would append a pair of lists, `Front` and `Rear`, given a pointer

to the head of each. The lists are composed of singly linked nodes, with a known, specially designated "`nil`" node marking the termination of each list. A straightforward approach would be to step down links in `Front` until reaching `nil`. At this point the *previous* node's link pointer, which originally pointed to `nil`, should be destructively modified to point to the head of `Rear` instead. Thus, as we step down `Front`, we must always keep track of the last node we traversed, as each successive node might be the final node before the end of the list.

The Prolog code for `append`/3 transformed to use Update in Place implements just this algorithm. First, however, we must define a means for performing basic destructive assignment. We use the primitive `setarg`/3, which takes the following form:

```
setarg( +ArgNum, +CompoundTerm, ?NewArgument ).
```

A call of this form replaces the `ArgNum`'th argument of `CompoundTerm` with the value of `NewArgument`. The interpretation of the arguments is analogous to that of `arg`/3. Furthermore, `setarg`/3 is backtrackable; a trail test is performed, and if necessary the old value of `CompoundTerm`'s `ArgNum`'th field is recorded on the trail so that backtracking can restore the original value later. Note that `setarg`/3 is provided as a primitive by several Prolog implementations [11, 12, 13].

Given this assignment primitive, the UIP transformed code for `append`/3 is as follows:

```
append( [], Rear, Rear ).
```

```
append( Front, Rear, Front ) :-
    Front = [ _ | Tail ],
    append_1( Tail, Rear, Front ).



append_1( [], Rear, Last ) :-
    setarg( 2, Last, Rear ).

append_1( Front, Rear, _ ) :-
    Front = [ _ | Tail ],
    append_1( Tail, Rear, Front ).
```

We notice immediately that the transformed code contains not two clauses, but four, half belonging to `append`/3 and half to `append_1`/3. Within each predicate, the first and second clauses were generated from the first and second clauses of the original code, respectively. We call `append`/3 an *entry predicate*, and `append_1`/3 an *internal predicate*. Since each clause from the original code appears as both an entry and internal clause, we may occasionally refer to the entry or internal *version* of a clause. In general, entry predicates have the same name as the original predicate, and are called using the same arguments. Internal predicates have modified arguments that track last references, and are called by each other and by entry versions. Internal predicates are named after the original predicate plus an additional suffix, "`_1`" in this case. The suffixes themselves encode additional information which is further explained below.

Returning to our example, the first clause of the entry predicate handles the specific case where the top level call to `append`/3 had the empty list as its first argument. In this case, the output is simply the second argument; this clause is identical to the corresponding

clause in the original code.

The second clause of the entry predicate is more interesting. If the first argument, `Front`, is not the empty list, then we reuse its first cell to make the first cell in the output list. Thus, head unification causes the third argument to be a reference to the same cell as the first. In the body, the unification grabs a reference to `Tail`, the next cell in sequence. Finally, the clause calls the internal predicate, `append_1`/3. In this call, `Tail` replaces the first argument and `Rear` is passed along unchanged, just as in the original code. However, in place of an output argument, we pass `Front` as a last reference. Should `Tail` be the empty list, it is `Front`'s second field that should be destructively changed to point to `Rear`.

Beyond this initial cell, it is the internal predicate `append_1`/3 that handles the work of repeatedly stepping forward down the first argument, always maintaining a reference to the last cell traversed. The second clause of `append_1`/3 recursively walks down the list. At each level of recursion, the unification gives us a reference to the next cell in the list, and the current cell is passed as a last reference to the recursive subgoal. This second clause is only executed when the first argument is not the empty list. This implies that the last reference it had received from its caller will not need to be modified after all. Thus, it is matched with "_" in the head and discarded. Where CTGC would have already put work into preparing that referenced cell for assignment, we have done no such extra work.

The recursive traversal ends with the first clause of `append_1`/3. Here, we have reached the end of the first list. The second argument, `Rear` still holds the list to be concatenated, and now `Last` references the cell just *before* (i.e. containing a reference to) the terminating `[]`. Upon entry, then, the empty list in the first argument is the value referenced by the tail field of `Last`. The call to `setarg`/3 destructively reassigns this field to point to `Rear`

instead. The second list has now been pasted onto the end of the first; the execution is complete. Where the original code would have copied as many cells as the entire length of the first argument, we have copied none. A single destructive assignment redirects the last cell of the first list to point to the head of the second.

2.2.2 Positional Arguments

In the call to `setarg`/3, notice that the number of the field to be changed is hard-coded as two. This works for `append`/3, as we are always modifying only that field. This may not be the case for other predicates, though. Consider the following naïve code for performing insertion into a binary tree:

```
tree_insert( Info, leaf, tree( Info, leaf, leaf ) ).

tree_insert( Info, tree( Node, Left, Right ), tree( Node, NewLeft, Right ) ) :-
    Info < Node,
    tree_insert( Info, Left, NewLeft ).

tree_insert( Info, tree( Node, Left, Right ), tree( Node, Left, NewRight ) ) :-
    Info >= Node,
    tree_insert( Info, Right, NewRight ).
```

Let us assume that static analysis has determined that the second argument is always dead following calls to `tree_insert`/3, and that the third argument is always a free variable at the time of call. We will update the second argument in place to construct the third.

Intuitively, one expects that the transformed code for the first clause will perform a destructive assignment to the parent of the `leaf` atom in the second argument to the first

clause, giving it a reference to a new tree node containing `Info` instead. However, `leaf` may have been either the left or right subtree of its parent. Hence, the destructive assignment might need to be made to either the second or third field of the last reference. One way to track which field should receive the assignment is to augment the internal predicates of clauses with an extra parameter. Whenever a `Last` is passed down to a body goal, an additional argument, `Pos`, accompanies it. `Pos` specifies which field of `Last` should be replaced in the event that destructive assignment is actually performed. The full "last reference," then, is a particular field (given by `Pos`) of a particular cell (given by `Last`).

The following is the full transformed UIP code for `tree_insert`/3:

```
tree_insert( Info, leaf, tree( Info, leaf, leaf ) ).

tree_insert( Info, Tree, Tree ) :-
    Tree = tree( Node, Left, _ ),
    Info < Node,
    tree_insert_2( Info, Left, Tree, 2 ).

tree_insert( Info, Tree, Tree ) :-
    Tree = tree( Node, _, Right ),
    Info >= Node,
    tree_insert_2( Info, Right, Tree, 3 ).



tree_insert_2( Info, leaf, Last, Pos ) :-
    setarg( Pos, Last, tree( Info, leaf, leaf ) ).

tree_insert_2( Info, Tree, _, _ ) :-
    Tree = tree( Node, Left, _ ),
    Info < Node,
    tree_insert_2( Info, Left, Tree, 2 ).

tree_insert_2( Info, Tree, _, _ ) :-
```

```
Tree = tree( Node, _, Right ),
Info >= Node,
tree_insert_2( Info, Right, Tree, 3 ).
```

The inner workings of this code are very similar to those of the transformed `append`/3, although now the second argument is traversed instead of the first. The internal version of the first clause of the original code contains the anticipated call to `setarg`/3. However, instead of assigning into a predetermined field of `Last`, we assign into the field specified by `Pos`. Examining the other internal and entry clauses, we see that `Pos` is passed as either two or three, depending on whether `Info` should be plugged in as the left or right subtree of the node under examination.

2.2.3 Additional Internal Predicates

The reader may have noticed that the internal predicates for `tree_insert`/3 use the suffix "`_2`" instead of "`_1`". The number chosen indicates which input argument is a child of the last reference upon entry to that predicate. This information is significant, as it may determine whether or not a clause must perform a destructive assignment to alter a last reference's relationship to the clause's input arguments. In `append`/3, `Last` is always the parent of the first argument; in `tree_insert`/3, `Last` is always the parent of the second argument. However, there may be occasions where the last reference's children can vary among different calls. If more than one input argument is being traversed, such as during a merge operation, the last reference may be a parent of either input argument. Another possibility is that *none* of the input arguments is an immediate child of the last reference. This can happen when splicing occurs, and certain cells in the input are skipped over without

incorporating them into the output. To reflect this situation, we use the suffix "_stray".

Transforming the following code for delete/3 demonstrates how suffixes vary depending on the parent/child relationship between last references and input arguments. The predicate removes all instances of its first argument from the list given as its second argument, placing the "cleaned" list in its third argument. First, the naïve implementation:

```
delete( Junk, [ Junk | OldTail ], New ) :-
    delete( Junk, OldTail, New ).

delete( Junk, [ Keep | OldTail ], [ Keep | NewTail ] ) :-
    Keep \== Junk,
    delete( Junk, OldTail, NewTail ).

delete( _, [], [] ).
```

One may think of delete as operating in one of two modes: a "skipping" mode and a "copying" mode. In the skipping mode, cells that match Junk are discarded and the output list being composed remains unchanged. In the copying mode, cells that do not match Junk are copied into the output list. These two modes correspond to the first two clauses listed above.

The UIP code for delete/3 will use destructive assignments to splice out those cells that would be dropped in the naïve code's skipping mode. Figure 1 illustrates visually how certain cell's successor references will be updated in place to skip over unwanted Junk cells.

Because of the possibility of skipping certain cells and recycling others, the UIP code for delete/3 must keep track of more information than just the last reference. In particular,

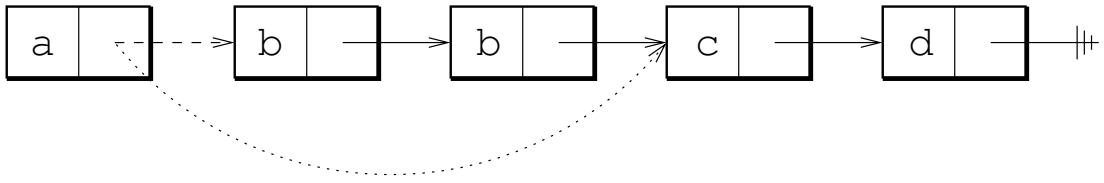

Figure 1: splicing in `delete`/3. This represents the behavior of UIP transformed code deleting all `b`'s from the list `[ a, b, b, c, d ]`. Solid arrows represent references from one list cell to the next that will not be changed. The reference represented by the dashed arrow from `a` to `b` will be destructively updated to reference `c` instead, as indicated by the lightly dotted arrow.

it must be able to detect *transitions* between the skipping and copying modes. Consider:

- When copying stops and skipping begins, the last reference when copying ended should be carried over the sequence of skipped cells.
- When skipping stops and copying begins, the last reference that had been carried across the skipped cells should be destructively modified to point to the first cell in the coming sequence of copied cells.

To return to Figure 1, upon visiting the first cell containing `b` we transition from copying to skipping. As the two `b` cells are skipped, we continue to pass down the second field of cell `a` as the last reference. Cell `c` prompts a transition back to copying mode, so we modify cell `a`'s second field (the last reference) to point to cell `c`.

The transition from skipping to copying is one of two points where the transformed `delete`/3 will have to make assignments. The other arises if we are still skipping when we reach the end of the list. In this case, the cell containing the last reference should become the final cell in the output list. Destructively assigning the last reference to point to nil (`[]`) terminates the output list and the call is complete.

At any level of recursion, the critical factor in distinguishing copying and skipping modes rests in the location of the last reference relative to the input arguments. If the last reference is an immediate parent of one of the input arguments, then we are copying. If none of the input arguments is a child of the last reference, then we call this a "stray" and it signals skipping. These cases match the description given earlier of the meaning of the suffixes of internal predicates. In fact, tracking which input argument, if any, is a child of the last reference provides sufficient information for all transformed code to behave correctly under any pattern of discarding and reuse.

For `delete`/3, both the first and second arguments are considered input, but only the second argument is reused. At any moment, then, the last reference is either the parent of the second argument or of no input argument. The internal predicates, then, will be named `delete_2` and `delete_stray`.

The UIP-transformed code for `delete`/3 is as follows:

```
delete( Junk, [ Junk | OldTail ], New ) :-
    delete( Junk, OldTail, New ).

delete( Junk, Old, Old ) :-
    Old = [ Keep | OldTail ],
    Junk \== Keep,
    delete_2( Junk, OldTail, Old, 2 ).

delete( _, [], [] ).


delete_2( Junk, [ Junk | OldTail ], Last, Pos ) :-
    delete_stray( Junk, OldTail, Last, Pos ).
```

```
delete_2( Junk, Old, _, _ ) :-
    Old = [ Keep | OldTail ],
    Junk \== Keep,
    delete_2( Junk, OldTail, Old, 2 ).

delete_2( _, [], _, _ ).



delete_stray( Junk, [ Junk | OldTail ], Last, Pos ) :-
    delete_stray( Junk, OldTail, Last, Pos ).

delete_stray( Junk, Old, Last, Pos ) :-
    Old = [ Keep | OldTail ],
    Junk \== Keep,
    setarg( Pos, Last, Old ),
    delete_2( Junk, OldTail, Old, 2 ).

delete_stray( _, [], Last, Pos ) :-
    setarg( Pos, Last, [] ).
```

Notice that at the start of a top level call, the entry predicate skips over any initial `Junk` cells without dropping into an internal predicate. The main purpose of internal predicates is to keep track of last references and their relationships to input arguments. Also recall that the last reference is a field within the most recent input cell to be recycled into output. If the input list starts with `Junk` cells, then *no* cells will have been recycled into output until the first non-`Junk` cell is found. Until that point, there is no last reference, so execution remains inside the entry predicates.

The internal predicates for this example have arity four instead of three. As before, the extra argument specifies the field number within `Last` that contains the actual last reference. For `delete`/3 this is always two. In theory, it should be possible to optimize out this positional argument and place the constant two directly in the calls to `setarg`/3. For

the sake of simplicity, the present automated techniques for applying the UIP transformation do not attempt to detect this situation. However, Section 3.2.2 examines another assignment mechanism that allows the extra positional argument to be eliminated for all transformed predicates.

3 Implementation Issues

Gudjonsson [10] formally defines the code synthesis process in mathematical terms. This section takes up the primary intent of this paper: to carry that specification forward into implementation. Certain important implementation issues are omitted from the abstract specification, and the actual synthesizer may have several options as to how these details should be realized. In other areas, Gudjonsson's specification makes certain decisions for the sake of descriptive simplicity that might not actually lead to ideal transformed code.

The implementation questions that are left to the synthesizer's discretion fall into two major categories: those related to unravels (a special type of unification), and those related to destructive assignments. Issues concerning unravels are described in Section 3.1. Those concerning assignments are covered in Section 3.2. The hypotheses and inferences developed in these sections are evaluated in light of empirical data in Section 4.

3.1 Unravels

Unravels are unifications that the synthesizer adds to the bodies of transformed clauses. They serve several important purposes. One is to obtain references to substructures given a reference to a parent structure. For example, if `In` points to the start of a list, the unravel "`In = [ _ | Tail ]`" will cause `Tail` to be a reference to the tail of that list. Such a reference might be needed so that `Tail` may be passed as an argument in a body goal, or so that a recycled cell can be destructively assigned to incorporate `Tail`. Alternately, if a destructive assignment is scheduled for the second field of `In`, but that field's current value will be needed elsewhere, it must be obtained *before* the destructive assignment is

performed. In addition to obtaining references to extant portions of structures, unravels may be used to bind nonground terms to new static structures. By convention, the left hand side of an unravel unification is the variable being *unraveled*. The right hand side is called a *template*, as it serves as a template or outline of the structure being unraveled.

The formal specification of the UIP transformation uses unravels liberally, often redundantly. Furthermore, unravel templates may cause the unification to examine structure deeper than is really needed. Also, under certain conditions unravels may confound other important optimizations. Each of these potential hindrances may have a corresponding optimization or work around. The three subsections that follow describe these issues and possible solutions in greater detail. Section 4.2 evaluates each empirically.

3.1.1 Minimized Unravels

When unravels are used simply to obtain references to substructures, certain simplifications may be applied. When composing the template for such an unravel, underscores may be substituted for entire substructures for which no reference is needed. For example, suppose that a structure appears in the original program as "`f( g( a ), g( b ), g( c ) )`", that `In` is a reference to the root of this structure, and that references to the first and third children of `In` are needed. A naïve unravel would simply substitute variables in for the required subterms, giving an unravel of the form "`In = f( First, g( b ), Third )`". However, if it is known that `In` is already instantiated to the original static structure, there is no need to examine the entire substructure of its second child. A more efficient unravel would be "`In = f( First, _, Third )`". This may seem an obvious optimization, but it is important to distinguish those contexts in which it can and cannot safely be applied.

In particular, when an unravel is used for its second purpose, to create *new* structure, templates cannot be simplified with underscores.

Once a reference to a given substructure has been obtained, any subsequent unravels that are also intended to retrieve it may be omitted. The code synthesizer is able to track which substructures have been located by a prior unravel, and avoids doing the same work twice. This optimization may be thought of as a variant of common subexpression elimination. As the synthesizer is processing unravels, it records the location of every subterm for which a reference has been obtained. If a goal requires a reference to this same subterm, no unravel is performed, and the reference already available is used. In many cases, avoiding redundancy can eliminate all but a small handful of unravel unifications in the transformed program. One final, extremely basic unravel minimization is possible: when the template in an unravel is itself a free variable or a simple atom, it may be unified with the variable being unraveled at synthesis time, causing the two to be expressed identically in the transformed clause. The trivial variable-to-variable or variable-to-atom unification need never appear as a goal in the body of the transformed clause.

3.1.2 Unravel Placement

The minimization techniques outlined above are able to significantly reduce the number of unravels that must appear in transformed programs. However, for those few that remain, the code synthesizer must choose exactly *where* in the transformed body the unravel unifications should be placed. One option, the one given in [10], is to perform all unravels at the start of a clause. However, this may lead the clause to perform unnecessary work: if some body goal fails, any references obtained for use by later goals will never be needed. Prolog clauses are

often informally structured so that early goals express guard conditions. Thus, failures are most likely early in a clause, and the transformed code should do as little work as possible while failure and backtracking is still a strong possibility.

An attractive alternative is to perform unravels at need. Unifications that fetch references to substructures should appear just before those substructures are actually needed, and no earlier. Similarly, unifications that bind variables to static structure should appear immediately before those body goals that use the variables being unraveled. We hypothesize that late placement of unravels will not hurt performance when failures are rare, and that it may improve performance when failures occur frequently. Postponing unravels may lend the additional benefit of reducing the number of live references that need to be preserved on the stack across subcalls, although we will not directly address this issue here.

3.1.3 Input Aliasing

The third unravel issue under examination concerns the effect that the UIP transformation has upon another major optimization: determinacy analysis. By examining the static structure of arguments in clause heads, the aggressively optimizing Aquarius Prolog Compiler is able to determine when certain clauses within a given predicate are mutually exclusive [14]. In such cases, execution is deterministic and Aquarius may be able to avoid creating choice points. This keeps both the trail and the stack small and reduces overhead associated with maintaining them. In `append`/3, for example, one head has nil as its first argument, while the other has a list cell. Assuming that all calls have their first argument ground, these two clauses are mutually exclusive: if the arguments passed during a given call match one clause, they cannot match the other. Therefore, no backtracking can occur across this point, and

no choice point need be created.

The conflict arises when code has already been transformed to perform UIP before Aquarius's analyses begin. The topmost memory cell of an input argument tends to be reused to construct output, so we will generally need a reference to it. According the the formal specification, the synthesizer should simply replace any static structure in the clause head with a simple variable, and unify that variable with the original static structure early in the clause body through a series of unravels. Consider the following example of a head before transformation, noting that underscores have been substituted for arguments not central to the present discussion:

```
take_first( [ [ Head | Tail ] | Lists ], _, _ ) :-
    ...
```

Assume that the first argument, a list of lists, is an input argument. If a reference is needed to the root of this argument, the variable that will hold the reference must appear in the head. Thus, the transformed clause would begin as follows:

```
take_first( In, _, _ ) :-
    In = [ [ Head | Tail ] | Lists ],
    ...
```

Unfortunately, moving these unifications from the head into the body often confounds Aquarius's determinacy analysis. The analysis is strongest when mutually exclusive unifications appear in heads; when those unifications instead appear in bodies, the determinism

is still present, but Aquarius is not always able to take advantage of it. We anticipate that blocking this important optimization will severely damage the performance of transformed code.

An alternative, input aliasing, keeps static structure in clause heads, but also allows us to obtain references to the root cells of these input arguments. Input arguments appear in the transformed head just as they do in the original, including full static structure. However, for each input argument, we add an additional argument, called an *input alias*, that appears as a simple variable in transformed clause heads. In internal calls to the transformed predicate, the input argument and the alias are passed identical values. During head unification, the input alias becomes a reference to the root of the input argument, and the original static structure is already available without requiring additional unravels in the body. Moreover, the static structure still appears in the head, so Aquarius's determinacy analysis continues to function properly. Using input aliasing, the example from above would appear thusly:

```
take_first( [ [ Head | Tail ] | Lists ], _, _, In ) :-
    ...
```

Calls to the original `take_first`/3 are modified to use the new `take_first`/4 instead. An original goal of the form `take_first( Lists, Heads, Tails )` would be replaced by `take_first( Lists, Heads, Tails, Lists )`.

This work around has the disadvantage of increasing the arity of transformed clauses, which may slow the calling process. However, we conjecture that this slowdown will be extremely small compared to the cost of breaking Aquarius's determinacy analysis.

3.2 Destructive Assignments

The second major cluster of implementation issues involve destructive assignment directly. Gudjonsson's formal specification adopts a simple mechanism for adding destructive assignments to clause bodies, and for implementing the destructive assignment primitive itself. A more sophisticated placement scheme may lead to more efficient code; similarly, several options exist for how to implement the destructive assignment primitive, and some may execute faster than others. The two subsections that follow describe these two destructive assignment issues in further detail, and Section 4.3 evaluates the implications of each empirically.

3.2.1 Upper Connection Placement

Just as in the case of unravels, there exists a certain amount of flexibility as to exactly where in a clause body destructive assignments should be performed. Loosely speaking, destructive assignments "piece together" new memory structures out of old ones. When these new structures are to be used in a body goal, the necessary assignments should be performed just before the call. Any earlier, and the "pieces" from which the structure is composed may not have been fully processed by earlier goals; in particular, the interactions between unravels and destructive assignments impose a partial ordering that must be obeyed to preserve proper semantics. Any later, and the goal itself will receive incorrectly formed structures as its arguments. However, when destructive assignments are used to assign to a last reference, we have greater leeway as to when the assignment should be done. We can choose to perform these assignments, called *upper connections*, anywhere in the body after the proper unravels have been performed. As in the case of unravels, it would seem wise

to postpone assignments as long as possible, so that less work need be backtracked in the event that a body goal fails.

However, postponing assignments too long may create other problems. Specifically, if assignments are pushed back so far that they become the last goals in the transformed clause, any tail-recursion or last-call optimization that might have been applicable to the original clause will no longer be available. As in the case of input aliasing, the synthesizer must not confound other important optimizations in the process of applying its own. Our hypothesis is that placing movable assignments just before the last subgoal in a clause body will give the best performance. For example, placing assignments late would result in the following code, drawn from the `split`/3 benchmark described below:

```
split_stray_1( [ _, _ | Tail ], Last1, Last2, In, Pos1, Pos2 ) :-
    In = [ _ | Second ],
    split_stray_1( Tail, In, Second, Tail, 2, 2 ),
    setarg( Pos1, Last1, In ),
    setarg( Pos2, Last2, Second ).
```

Notice that both of the `setarg`/3 calls perform assignments to last references. These assignments could just as well be placed before the recursive subgoal, giving the following:

```
split_stray_1( [ _, _ | Tail ], Last1, Last2, In, Pos1, Pos2 ) :-
    In = [ _ | Second ],
    setarg( Pos1, Last1, In ),
    setarg( Pos2, Last2, Second ),
    split_stray_1( Tail, In, Second, Tail, 2, 2 ).
```

While both of these clauses are semantically equivalent, the second is amenable to tail-recursion optimization while the first is not. Needless to say, we anticipate that empirical evaluation will show the second clause to execute far more rapidly.

3.2.2 Destructive Update Method

Beyond where to place destructive assignments, the exact method by which the assignments are performed is an interesting issue. The examples introduced earlier have used `setarg`/3. This is a simple, straightforward approach, and makes a source-to-source transformation reasonably portable. However, `setarg`/3 may be too general a tool to perform the required task efficiently. For one, `setarg`/3 may be applied to any type of memory cell, but the work that it needs to do may change depending on its actual run time arguments. Specifically, consider a call of the following form:

```
setarg( +ArgNum, +CompoundTerm, ?NewArgument ).
```

If `CompoundTerm` is a cell with the standard memory layout, the memory location to be modified is offset from the start of the cell by `ArgNum` words, as the first word (offset zero) holds `CompoundTerm`'s functor. However, most Prolog implementations optimize storage for list cells, storing them in only two words without reserving the first word for a functor. Thus, if `CompoundTerm` is a list cell, the memory location to be modified is offset by only `ArgNum` $-$ 1 words. Unless compile time analyses include aggressive type propagation down to the level of builtins, `setarg`/3 will have to perform tag checks and conditionally an integer decrement for every call. Figure 2 illustrates in a simplified manner the way in

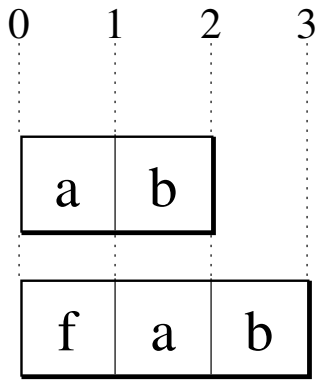

Figure 2: The upper rectangle represents the memory layout of the list cell `[ a | b ]`. The lower rectangle represents the memory layout of the structure cell `f( a, b )`. Notice that the first and second arguments of each, `a` and `b`, are located one word further back in the list cell than in the structure cell.

which list cells and standard cells place the same fields at different offsets.

An alternative would be to propagate type information during UIP code synthesis, and invoke one of three new, specialized destructive assignment primitives. These three new primitives are of the following forms:

```
assign_general( +ArgNum, +CompoundTerm, ?NewArgument ).

assign_struct( +ArgNum, +Structure, ?NewArgument ).

assign_list( +ArgNum, +ListCell, ?NewArgument ).
```

The first of these, `assign_general`/3, performs the same task as `setarg`/3. It makes no assumptions about the type of `CompoundTerm`. However, `assign_general`/3 never decrements `ArgNum` to compensate for the missing functor field of a list cell, unlike `setarg`/3. Rather, at code synthesis time, when a new last reference is being passed to a body goal and it is known to represent a list cell, the positional parameter ("`Pos`") is decremented

by the synthesizer before it appears in the transformed code. This primitive will only be used when more detailed type information is unavailable, such as when either a list or standard cell might reach the same destructive assignment point. Memory cells that are known to contain structures using the standard memory layout are destructively modified using `assign_struct`/3. This predicate performs no tag checking and always modifies the word at offset `ArgNum` within `Structure`. Lastly, `assign_list`/3 is used to modify known list cells. It also performs no tag checking and always modifies the word at offset `ArgNum` within `ListCell`. As in the case of `assign_general`/3, `assign_list`/3 does not decrement the positional parameter. That adjustment will have already been performed by the code synthesizer. The elimination of most run time tag checks and all run time decrements should allow transformed code to execute faster, although it is at the expense of greater dependence upon the underlying representation of memory cells.

Both of the destructive assignment mechanisms presented thus far require a last reference consisting of both a reference to a memory cell and a positional parameter representing either a field number or adjusted offset. However, given sufficient access to the underlying memory representation mechanisms, it is possible to condense the last reference into a single parameter. Consider: the only information that a last reference need convey is the location of the field that may need to be destructively updated. Instead of passing last references as a cell base reference and offset, we can instead pass a single direct pointer into the interior of the cell. When a clause needs to pass a last reference to a body goal, it obtains a reference to the proper memory cell, increments that reference so that it points directly at the proper field of the cell, changes its tag to be that of a free variable, and passes this new pointer as the last reference. At the point of destructive assignment, no offset need be specified, and an

`assign`/2 primitive performs a simple assignment into memory. Thus, where a transformed program might formerly have contained the following code:

```
foo :-
    ..., bar( In, Last, 2 ), ...

bar( In, Last, Pos ) :-
    setarg( Pos, Last, In ),
    ...
```

Instead, using this increment-before-call scheme to eliminate positional arguments, the transformed code becomes:

```
foo :-
    ...,
    increment( Last, 2, Pointer ),
    bar( In, Pointer ),
    ...

bar( In, Last ) :-
    assign( Last, In ),
    ...
```

Here, the `increment`/3 primitive sets `Pointer` to point into the interior of `Last` at an offset of two words.[1] It may appear here that the increment scheme for performing assignments is falling into the same trap as CTGC: performing additional work in anticipation of a destructive assignment that may never be performed. However, close scrutiny of compiled

[1] In truth, no `increment`/3 primitive exists. Rather, a collection of primitives of the form `inc_`*type*`_`*n*/2 are used, where *type* is either `list` or `struct` and *n* varies from 1 to a reasonable maximum anticipated offset. This is an artifact of the manner in which primitives are added to Aquarius as Berkeley Abstract Machine macros. By "hard wiring" the constant, significantly more efficient BAM code may be produced.

code that uses this approach reveals that the `increment`/3 primitive can usually be executed as a *single* register-to-register add operation. As a transfer of this type is otherwise used to load the positional argument into an argument register before a call, and an add operation takes one cycle, `increment`/3 does not result in a net slowdown on the calling end. We anticipate that at the point of destructive assignment, the extremely simple `assign`/2 primitive will be able to execute more rapidly than either `setarg`/3 or any of `assign_*`/3. Furthermore, reducing the arity of transformed predicates may help reduce register pressure, facilitating other optimizations. The empirical data that follow will reveal how significant these speed differences are. For now, we conjecture that `setarg`/3's generality will cause it to be the slowest destructive assigner, that `assign_*`/3 will be somewhat faster, and that the the fastest performance will be obtained with `increment`/3 and `assign`/2.

4 Empirical Evaluation

The implementation issues described above are difficult to resolve with confidence unless real, demonstrable variations in execution speed may be found. This section examines the results of timing each of a collection of seven benchmarks under the various conditions proposed in Section 3. In all cases, the code being tested was generated by a prototype UIP code synthesizer directly from naïve source, augmented by special guiding directives which are detailed in the Appendix.

The subsections that follow present, for each choice of implementation options, comparative timings of naïve code and different variations of UIP transformed code. Our primary goal is to determine which choices as to how to perform synthesis will result in the fastest overall code for most programs. All benchmarks were transformed using the prototype code synthesizer. Both naïve and transformed programs have been compiled using release 1.0 of the Aquarius Prolog Compiler [12, 14] with all analyses and optimizations enabled.

Time trials were conducted on an unloaded SPARCstation[2] IPC with 28Mb of RAM. The running environment for the benchmark executables was the standard provided by Aquarius, with the exception that the trail was increased to a size of 1.5Mb. Each executable is evoked repeatedly until twenty such executions complete with no page faults. System- and user-mode execution times as reported by `time` [15] are summed, and the single execution with the lowest total is reported. In general, times vary by no more than 0.1 second across all twenty trials.

Section 3 informally proposed which choices of synthesizer options we expect will produce

[2]SPARCstation is a registered trademark of Sun Microsystems, Inc.

the best code. In the tests that follow, while one choice is varied all others are held constant at the expected best values. Specifically, where not otherwise specified:

- unravels are minimized as much as possible.
- unravels are performed as late as possible.
- input aliasing is enabled.
- assignments are performed using `increment`/3 and `assign`/2.
- assignments for upper connections are placed as late as possible in clause bodies, but still before the last subgoal.

4.1 Benchmarks

The benchmark suite has been chosen to illustrate the performance of transformed programs in a variety of contexts. All of the programs in the benchmark suite are fairly simplistic, as certain earlier reuse analysis stages of the UIP transformation have not yet been automated. Nevertheless, the selected programs do adequately span most of the relevant issues. The programs upon which the UIP optimization's effectiveness will be evaluated are as follows:

append A list of length one is appended onto a list of length 500,000.

delete A single integer is located and deleted from a list in which it is preceded by 500,000 differing integers.

insert An integer is inserted into an ordered list of integers requiring 500,000 members of the list to be skipped before finding the proper point of insertion.

merge An ordered list of the first 70,000 even nonnegative integers is merged with an list of the first 70,000 odd nonnegative integers so that the resulting list is also ordered.

split A list of 500,000 integers is split so that every other member is in one of two output lists.

take A list of 200,000 lists is split into a list of the heads and a list of the tails of its members.

tree A node is added to a simple binary search tree requiring 200,000 nodes to be traversed from root to leaf.

Appendix B provides the full source code for each of these benchmarks, including the reuse directives that guide the code synthesizer.

4.2 Unravels

Section 3.1 described certain important implementation issues concerning the handling of unravels by the synthesizer, and attempted to predict which choices would result in the most efficient transformed programs. The three subsections that follow evaluate the accuracy of these predictions in light of empirical data drawn from the benchmark suite outlined above.

4.2.1 Minimized Unravels

To measure the impact of unravel minimization, all benchmarks were executed and timed first with unravels minimized as much as safely allowed, and then with no attention paid to unravel simplification. Table 1 presents the net execution times and time ratio of transformed to naïve code.

Benchmark	Naïve	Minimized		Unsimplified	
	Net Time	Net Time	Ratio	Net Time	Ratio
append	1.5	0.5	3.0	0.6	2.5
delete	1.9	0.8	2.3	1.5	1.2
insert	1.9	0.8	2.3	2.1	0.9
merge	1.0	0.3	3.3	0.7	1.4
split	1.6	0.7	2.2	1.3	1.2
take	1.4	0.7	2.0	1.1	1.2
tree	1.3	0.5	2.6	1.0	1.3

Table 1: Empirical comparison of benchmark speeds with unravels minimized versus not simplified at all. All times in all tables have been adjusted to reflect only update time; the constant time required to construct the original input structures has been subtracted. Net times are reported in seconds; ratios are the quotient of the net time for the naïve implementation and the net time for the transformed program.

One immediately sees that minimizing the number and complexity of unravels is a significant optimization. With unravels minimized, UIP code executes two to more than three times faster than naïve code. Without this aggressive simplification, the fastest benchmark was only 2.5 times faster than naïve code, and `insert` actually executed ten percent more slowly. Examination of the synthesized code for `insert` reveals that when unravel minimization is used, *all* explicit unifications for unravels are removed from the transformed program. Without minimization, sixteen unifications result, with as many as three appearing in a single clause. This is extra run time work that could just as well be eliminated at compile or synthesis time. On average, benchmarks executed nearly two times faster with unravels minimized than without.

4.2.2 Unravel Placement

The current benchmark suite does not satisfactorily explore the question of how placement of unravels in clause bodies affects performance. As noted earlier, this distinction should

Benchmark	Naïve	At-Need		Early	
	Net Time	Net Time	Ratio	Net Time	Ratio
append	1.5	0.5	3.0	0.6	2.5
delete	1.9	0.8	2.3	0.8	2.3
insert	1.9	0.8	2.3	0.8	2.3
merge	1.0	0.3	3.3	0.3	3.3
split	1.6	0.7	2.2	0.7	2.2
take	1.4	0.7	2.0	0.7	2.0
tree	1.3	0.5	2.6	0.5	2.6

Table 2: Empirical comparison of benchmark speeds with unravels placed in clause bodies just before need versus as early as possible.

be most significant when backtracking occurs frequently. The benchmarks used here do not backtrack. Therefore, one expects that performing unravels only at need will not significantly improve performance. Table 4 presents timing data that confirm this expectation.

All benchmarks executed identically regardless of the positioning of unravels. In fact, for several benchmarks, the actual UIP code produced is identical. For `insert`, for example, early or at-need placement is a meaningless distinction since no unravels actually appear in the synthesized code. An important area for future exploration would be the behavior of more complex benchmarks that contain significant unravels and that do backtrack frequently. The data presented here, however, do establish an important result: in code that rarely backtracks, placement of unravels is not significant; any safe placement may be used without adversely affecting performance.

4.2.3 Input Aliasing

Input aliasing is not an optimization *per se*, but rather a means to avoid blocking Aquarius's determinacy analysis. It is unfortunate that removing static structure from clause heads

Benchmark	Naïve	Aliasing		No Aliasing	
	Net Time	Net Time	Ratio	Net Time	Ratio
append	1.5	0.5	3.0	0.4	3.7
delete	1.9	0.8	2.3	3.0	0.6
insert	1.9	0.8	2.3	3.0	0.6
merge	1.0	0.3	3.3	9.0	0.1
split	1.6	0.7	2.2	1.7	0.9
take	1.4	0.7	2.0	0.6	2.3

Table 3: Empirical comparison of benchmark speeds with input arguments aliased versus no aliasing.

can confound this analysis, but the optimizations that determinacy detection permits are too valuable to ignore. Table 3 contains the timing results for all benchmarks save `tree`, which suffered such reduced efficiency without input aliasing as to be unable to execute on inputs of the required test size.

Even those benchmarks that were still able to process their test data endure considerable speed reductions without benefit of Aquarius's determinacy analysis. The hardest hit of the benchmarks, `tree` excepted, is `merge`, which executes ten times more slowly than naïve code unless input arguments are aliased. The two exceptions to this slowdown are `append` and `take`, which executed slightly faster without aliases. It would appear that Aquarius is still able to detect determinism for these two benchmarks. This is confirmed by the fact that `append` and `take` compile down to BAM code containing no choice points regardless of whether aliasing is used. By contrast, all other benchmarks contained more choice points when compiled with aliasing disabled than with aliasing enabled. The reason why Aquarius retains the ability to detect determinism only in `append` and `take` remains somewhat unclear.

4.3 Destructive Assignments

The heart of the UIP transformation is destructive assignment. While destructive assignment is conceptually simple, the synthesizer has several options concerning how best to implement and use this operation. Section 3.2 discussed these issues and conjectured how their resolution would affect the efficiency of transformed programs. The two subsections that follow evaluate these conjectures against the performance of the benchmark suite outlined earlier.

4.3.1 Upper Connection Placement

As in the evaluation of the impact of unravel placement, the current benchmark suite cannot fully explore the issue of ideal upper connection placement. Because backtracking never occurs, we expect that placing unravels early or later in a body will have no effect on performance. However, as predicted in Section 3.2, if assignments for upper connections are placed absolutely last in a body, they may prevent tail-recursion and last-call optimizations. Table 4 contains time trial data that confirms these predictions.

In all cases, placing upper connections just after the guard goals of transformed bodies or just before the last subgoal did not affect execution.[3] This is encouraging, as placing assignments later in a body may lead to speedups when backtracking is frequent. The results given here verify that doing so will not hinder forward-executing code. See [9] for a preliminary investigation of the performance of UIP transformed code in a heavily backtracking environment.

[3]The synthesizer uses a simple but quite effective definition of *guard goals*. The guards of a clause are defined to be those goals that appear at the start of the body and which do not lead to subcalls in the compiled code. Thus, $</2$ would be a guard but `write`/1 would not.

Benchmark	Naïve	Before Tail		Early		Late	
	Net Time	Net Time	Ratio	Net Time	Ratio	Net Time	Ratio
append	1.5	0.5	3.0	0.5	3.0	0.5	3.0
delete	1.9	0.8	2.3	0.8	2.3	0.8	2.3
insert	1.9	0.8	2.3	0.8	2.3	0.8	2.3
merge	1.0	0.3	3.3	0.3	3.3	6.7	0.1
split	1.6	0.7	2.2	0.7	2.2	-	-
take	1.4	0.7	2.0	0.7	2.0	-	-
tree	1.3	0.5	2.6	0.5	2.6	0.5	2.6

Table 4: Empirical comparison of benchmark speeds with upper connection assignments placed in clause bodies just before the last subgoal versus early versus late. Late placement rendered `split` and `take` non-viable.

The times listed for tests that use late placement highlight the need to avoid confounding tail-recursion and last-call optimization. Three of the benchmarks, `append`, `insert`, and `tree`, do not suffer slowdowns as they contain destructive assignments only in contexts where no recursive subgoals are found. One benchmark, `delete`, could potentially suffer, as in one clause tail recursion becomes hidden by a late assignment. However, this code path is not crossed often enough when using the benchmark's input data to cause a visible reduction in speed. The three remaining benchmarks, `merge`, `split`, and `take`, suffer great efficiency losses when assignments are placed too late; `merge` runs ten times more slowly than naïve code, and `split` and `take` were not even able to complete execution without running out of stack space.

4.3.2 Destructive Update Method

Three options have been presented for performing destructive assignments. The members of the benchmark suite were timed using each, and these results may be found in Table 5.

The empirical data confirms our hypothesis that `setarg`/3's generality hampers its per-

Benchmark	Naïve	Inc/Assign		Assign		Setarg	
	Net Time	Net Time	Ratio	Net Time	Ratio	Net Time	Ratio
append	1.5	0.5	3.0	0.5	3.0	0.5	3.0
delete	1.9	0.8	2.3	0.8	2.3	0.8	2.3
insert	1.9	0.8	2.3	0.8	2.3	0.8	2.3
merge	1.0	0.3	3.3	0.3	3.3	0.8	1.2
split	1.6	0.7	2.2	0.8	2.0	2.3	0.6
take	1.4	0.7	2.0	0.7	2.0	1.8	0.7
tree	1.3	0.5	2.6	0.5	2.6	0.5	2.6

Table 5: Empirical comparison of benchmark speeds with assignments performed via increment and assign versus position passing and assign versus position passing and setarg.

formance, and that using early increments and thereby eliminating the positional parameter results in the fastest code. The benchmarks that are of particular interest are `merge`, `split`, and `take`. These three benchmarks are the most assignment intensive, performing a destructive assignment at every level of recursion. The `merge` benchmark using `setarg`/3 was barely able to outpace naïve code; both `split` and `take` actually ran considerably slower. Using a positional parameter and `assign_*`/3 allowed these programs to outstrip the naïve implementation by respectable distances. Finally, `split` was still faster, albeit only slightly, using `increment`/3 and `assign`/2 to eliminate the positional parameter.

This result is extremely encouraging because of its implications regarding the applicability of UIP to ill-suited programs. Even when destructive assignments would need to be performed with great frequency, the UIP transformation still yields significant improvements over naïve code. Of course, this does require an efficiently implemented set of destructive assignment primitives. When assignment is too expensive, such as when performed by `setarg`/3, the result may be a net loss in speed.

Benchmark	Naïve	UIP		CTGC
	Time	Best Ratio	Standard Ratio	Ratio
append	1.5	3.7	3.0	2.0
delete	1.9	2.3	2.3	1.5
insert	1.9	2.3	2.3	-
merge	1.0	3.3	3.3	2.4
split	1.6	2.2	2.2	-
take	1.4	2.3	2.0	-
tree	1.3	2.6	2.6	2.4

Table 6: Empirical comparison of execution times of naïve implementation versus best and standard UIP transformed programs versus CTGC transformed programs.

4.4 General Results

The choices made concerning each synthesizer implementation issue have a profound effect on the speed of the resulting code. Table 6 summaries the speedups obtained using the synthesizer options proposed as ideal and the best speedup possible using any combination of synthesizer options. In nearly all cases, the options that had been conjectured as preferable did indeed produce the fastest transformed programs. The only anomalies were `append` and `take`, which did not suffer reductions in speed when input argument aliasing was disabled. This table also includes time ratios for four of the same benchmarks transformed using CTGC. The CTGC timing data is reproduced from that found in [9].

Broadly speaking, the code synthesizer is able to produce UIP transformed programs that update data structures two to nearly four times faster than naïve implementations. Although the peculiarities of each benchmark cause them to react differently to choices made for the synthesizer, the set of options conjectured as most generally desirable does give excellent results throughout. As predicted, the extra work performed by CTGC transformed programs consistently limits their speed. For the given benchmarks, UIP transformed code

Benchmark	Naïve	UIP	
	Net Memory	Net Memory	Ratio
append	9344	16	489.0
delete	9368	72	108.5
insert	9368	64	122.1
merge	3728	40	109.2
split	9360	48	162.6
take	10920	0	∞
tree	7776	24	261.0

Table 7: Comparison of benchmark memory comsumption for naïve implementations versus UIP transformed programs. All counts have been adjusted to reflect only memory consumed during data structure update; the constant memory required to construct the original input structures has been subtracted. Net memory usages are reported in bytes; ratios are the quotient of the net consumption for the naïve implementation and the net time for the transformed program.

executes on average thirty seven percent faster than the corresponding CTGC code.

Although the paramount concern of most optimizations is to improve execution speed, the amount of memory that a program requires to run is also an important issue. By aggressively reusing memory, programs transformed by the UIP technique consume far less memory than their naïve counterparts. Table 7 compares the net memory consumed by each member of the benchmark suite.

The reduction in memory use demonstrated by programs transformed using UIP is remarkable. Whereas naïve implementations consumed from 7.5Kb to 10.5Mb to process their data, no UIP transformed benchmark required more than 72 additional bytes of storage. The `append` benchmark reduced its memory consumption by a factor of nearly five hundred, and `take` was able to complete execution without requiring a single additional byte of storage.

5 Conclusions

The Update in Place transformation has the potential to produce programs that run significantly faster than naïve implementations. UIP is able to avoid Compile Time Garbage Collection's pitfall of performing unnecessary extra work preparing for destructive assignments that are never performed. By augmenting the Prolog language with primitives for performing high speed backtrackable destructive assignment, the UIP transformation offers the promise of considerably enhanced performance. The benchmarks tested here execute two to more than three times faster following transformation, and require vastly less memory. Even in tests where destructive assignment is performed with great frequency, UIP transformed programs still execute more rapidly in less space than naïve implementations.

Earlier explorations of UIP have relied upon manual translation of source programs [16]. The prototype code synthesizer demonstrates that automating the transformation process is indeed feasible, and that automatically synthesized code can perform as well as that produced by hand. Certain important implementation issues arise in the process of prototyping the synthesizer, and empirical evaluation has confirmed intuitive notions as to how these issues may best be resolved. Most significantly, the UIP transformation must take care that the changes it introduces do not confound other optimizations such as determinacy analysis and tail-recursion optimization. Examination of unravel simplification reiterates the value of using transformation time analyses to minimize the amount of additional work that the synthesized code must do. Finally, the comparison of mechanisms for performing destructive assignment highlights some of the tradeoffs between speed and portability. Although the entire UIP optimization may be expressed as a high level source-to-source

translation, faster code results when assignment primitives sacrifice portability in order to take advantage of low level representation details.

Several important questions about the UIP transformation remain open to investigation. Foremost among these is the challenge of choosing how to reuse memory cells most efficiently in more complex contexts than the simple benchmarks used in this paper. The dataflow analysis stage, although further developed, is still relatively immature. Once these two stages have been more deeply investigated and prototype implementations developed, it should be possible to automatically apply the Update in Place transformation to general programs written in Prolog as well as other languages. The findings described in this paper are a strong statement that such an effort would prove greatly rewarding.

6 Acknowledgments

This research has been funded in part by the National Science Foundation, through a Research Experiences for Undergraduates supplement to CISE grant CDA 8914587.

A Using the Code Synthesizer

A.1 Portability

This appendix is intended as an informal user's guide to the prototype Update in Place Code Transformer. The code synthesizer itself has been written to be as portable as is reasonably possible, and to the author's best knowledge conforms to the ISO Draft Standard as of the date of this document's printing [17]. Currently, implementations of the synthesizer exist for and have been tested in the following Prolog environments:

- SICStus Prolog release 0.7, compiled
- SICStus Prolog release 0.7, interpreted
- Aquarius Prolog release 1.0, compiled
- Aquarius Prolog release 1.0, interpreted

Output from the synthesizer is portable only to the extent that the additional primitives it uses are available across platforms. When `setarg`/3 is used as the vehicle for performing destructive assignment, any Prolog implementation that provides this as a built in should be able to execute UIP transformed code [11, 12, 13]. The other primitives described in Section 3.2 have currently been implemented only as Berkeley Abstract Machine macros for use with the Aquarius compiler.

A.2 Invoking the Synthesizer

The synthesizer provides two front end interfaces. When the synthesizer has been compiled down to a free standing executable, it may be invoked from the command line as follows,

assuming that the executable is named `siva`:

```
siva {<filenames> <option settings>}
```

File names and option settings may be freely interspersed. Once set, a given option affects the processing of all files that follow it on the command line unless overridden elsewhere. Further details on what options are available and the means by which they may be set are provided in Appendix A.4. Filenames should be fully qualified; no "`.pl`" or other suffix is assumed. If the filename "`user`" is given, the synthesizer will read from the standard input stream instead of a physical file. All output is sent to the standard output stream, where it may be redirected for storage into a physical file at the user's option. If no arguments are given, the set of default option values is simply printed.

If the synthesizer is being used in an interpreted environment, or is being called directly by other Prolog code, the `go`/1 and `go`/0 predicates provide a simple interface. The former will append the suffix "`.pl`" onto its argument, open a file with the resulting name, and transform the contents of that file. The later, which takes no arguments, simply reads from the current input stream, transforming what it receives.

A.3 Input File Format

Input files to the code synthesizer consist of standard Prolog programs plus directives that specify how cells are to be reused and how body goals of transformed clauses should be rewritten. Two directives provide the needed information: predicate reuse directives contain information about entire predicates, and clause reuse directives describe individual clauses of a single predicate.

A.3.1 Predicate Reuse

The predicate reuse directive has the following form:

```
:- pred_reuse( +PredName, +InputPosns, +OutputPosns, +Versions )
```

The four required arguments are interpreted as follows:

`PredName` identifies the predicate being described. It consists of the atomic name of the predicate and the predicate's arity, separated by a slash.

`InputPosns` is a list of positive integers identifying those argument positions of the predicate that correspond to input arguments. This list need not be ordered, but should contain no duplicates or out-of-range values. The list may be empty.

`OutputPosns` is a list of positive integers identifying those argument positions of the predicate that correspond to output arguments. This list need not be ordered, but should contain no duplicates or out-of-range values. The list may be empty. `OutputPosns` and `InputPosns` should be disjoint but need not include all arguments of the predicate.

`Versions` is a list of transformed versions of this predicate that the synthesizer should emit. Each member of the list should be one of the following:

- the atom `entry`

  An entry version consisting of a single clause with the same name and calling conventions of the original predicate will be produced.

- a list of last reference descriptors, exactly one for each member of `OutputPosns` and ordered to correspond with them.

  A last reference descriptor may be any one of the following:

- the atom orphan, indicating that no last reference yet exists for the corresponding output argument.
- a pair of the form `( +InPos, +Type )`, where `InPos` is an input argument position in `InputPosns` that is a child of this corresponding last reference, and `Type` is one of the atoms `list`, `term`, or `unknown`, depending upon the type of cell containing the last reference.
- a pair of the form `( stray, +Type )`, indicating that no input argument is a child of this corresponding last reference, and `Type` is one of the atoms `list`, `term`, or `unknown`, depending upon the type of cell containing the last reference.

Predicate reuse directives must appear *after* at least one clause for the corresponding predicate. Predicates that are not to be transformed need not supply a predicate reuse directive.

A.3.2 Clause Reuse

The clause reuse directive has the following form:

`:- clause_reuse( +ClauseName, +`β`, +G)`

The three required arguments are interpreted as follows:

`ClauseName` identifies the clause being described. It consists of the atomic name of the clause, the clause's arity, and the sequential order of this clause in relation to all clauses having the same name and arity. We will subsequently refer to the clause identified by `ClauseName` as the *current clause*.

Thus, the first and second clauses of `append`/3 would be named `append`/3/1 and `append`/3/2, respectively.

β is a list of pairs of paths to subterms in the clause being described. The pairs form a one-to-one mapping. Preimages are memory cells that are to be reused; postimages are structures that will be composed by reusing old cells. See below concerning the format of paths.

`G` is a list of pairs describing how last references in body goals should be formed. The first argument of each pair in `G` is a path to an output argument of a body goal that is to be transformed. See below concerning the format of paths. We will subsequently refer to the body goal containing the specified output argument as the *current subgoal*. The second argument of each pair in `G` is one of the following:

- the atom `orphan`, indicating that no last reference exists for the specified output argument.
- a term of the form `outside( +ParentLastNum )`, indicating that the last reference should be copied directly from one given in the head of the clause being transformed. `ParentLastNum` is a member of `OutputPosns` for the current clause, and the corresponding last reference is the one to be passed to the current subgoal.
- a term of the form `oldStray( +ParentPath, +ChildNum)`, indicating that the last reference should consist of the `ChildNum`'th field of the subterm located at `ParentPath`. The last reference is stray: it is not a parent of any of the input arguments to the current subgoal. Furthermore, the last reference is contained in

an "old" memory cell that has already been allocated upon entry to the current clause.

- a term of the form `newStray( +ParentPath, +ChildNum)`, which is interpreted in the same manner as `oldStray` except that the last reference is contained in a "new" memory cell that originally appeared as static structure in the current clause.
- a term of the form `old( +ParentPath, +ChildNum, +InPos)`, indicating that the last reference should consist of the `ChildNum`'th field of the subterm located at `ParentPath`. The last reference is not stray: `InPos` is a member of `InputPosns` for the current subgoal being constructed, and it indicates which input argument to the current subgoal is a child of the last reference. Furthermore, the last reference is contained in an "old" memory cell that has already been allocated upon entry to the current clause.
- a term of the form `new( +ParentPath, +ChildNum, +InPos)`, which is interpreted in the same manner as `old` except that the last reference is contained in a "new" memory cell that originally appeared as static structure in the current clause.

Clause reuse directives must appear at some point *after* the corresponding clause. Clauses of predicates that are not to be transformed need not supply a clause reuse directive.

For the sake of efficiency and ease of manipulation, the synthesizer expects and stores all paths as lists of zero or more positive integers stored in *reverse* order. For example,

given the term `expand( [], [ Head | Tail ], Others)`, the path `[ 1, 2 ]` represents the first child of the second child of the root, or `Head`.

All paths are rooted at the principal functor of the clause, with the clause being structured as any other term. Thus, for clauses with bodies, the principal functor is `:-`/2; the head is located at `[ 1 ]`, and the body consists of all paths of the form `[ ..., 2 ]`. For facts, or clauses with no bodies, the principal functor is the functor of the head itself, which would therefore be located at the path `[]`.

A.4 User Options

The code synthesizer supports several user options that may be used to determine how the implementation issues discussed in Section 3 should be resolved for the code being generated. The following options may be set at code synthesis time:

`unravel_depth` may be set to either `shallow` or `deep`.

> When set to `shallow` unravels will be performed as simply and rarely as possible, with trivial and redundant unravels being removed and templates simplified with underscores where possible. When set to `deep`, so such simplifications occur, and unravels appear as literally described in [10]. The default setting is `shallow`.

`unravel_placement` may be set to either `late` or `early`.

> When set to `late`, unravels are performed just before the goals that actually require them are executed. When set to `early`, all unravels that may be needed are performed at the start of the transformed clause's body. The default setting is `early`.

`input_aliasing` may be set to either `on` or `off`.

When set to `on`, arguments listed in `InputPosns` for this predicate will be aliased to aid determinacy analysis. When set to `off`, or if `InputPosns` is the empty list, no aliasing will be performed. The default setting is `on`.

`assign_placement` may be set to either `tail`, `early`, or `late`.

When set to `tail`, assignments for upper connections are placed immediately *before* the last body goal of a transformed clause. When set to `early`, upper connections are placed near the start of transformed bodies, after any "simple" opening goals that likely serve as guards. When set to `late`, upper connections are placed *after* the last body goal of a transformed clause. The default setting is `tail`.

`assign_method` may be set to either `increment`, `setarg`, or `assign`.

When set to `increment`, the `increment`/3 and `assign`/2 primitives are used. When set to `setarg`, the `setarg`/3 primitive is used. When set to `assign`, the `assign_*`/3 primitives are used. The default setting is `increment`.

Three mechanisms exist for modifying the settings of these user options. When the synthesizer is invoked from the command line, option names and values may simply be listed in and among file names. Thus, one possible invocation might be "siva unravel_depth deep foo.pl". Note that contrary to Unix[4] conventions, option names should not be preceded by a dash. Doing so conflicts with SICStus's option handling library.

Options may also be set directly in the input file to the synthesizer. A directive of the form "`:- siva_option( +Name, +Value )`" will set the named user option to the re-

[4]Unix is a registered trademark of Unix System Laboratories.

quested value. Lastly, if the synthesizer is in an interpreter environment, evaluating the goal "`siva_option( +Name, +Value ).`" will perform the same function.

Once set, options affect the synthesis of all files that follow. If an option is set using directives in the midst of an input file, those settings will affect all code synthesis for that file and all subsequent files. At this time, it is not possible to specify different values of a single synthesis option to be applied to individual clauses within the same file.

A.5 Known Limitations

The synthesizer uses an intermediate representation that requires augmenting clauses with `$old`/1 and `$new`/1 pseudo-goals. If such goals actually were present in the original clause, the synthesizer's behavior becomes ill determined.

The synthesizer must hold the entire source code in memory at once. The size of program that can be transformed, then, is limited by the amount of memory that can be allocated to the synthesizer.

Predicate and clause reuse directives are not checked to verify that they are reasonable. If errors are present in the reuse directives, the synthesizer will produce unexpected code or more likely fail.

As described, option settings affect entire files at a time, and cannot be varied among individual clauses or predicates that share a single file.

B Benchmark Source Code

The complete source code for the members of the benchmark suite is presented in this section, exactly as they are fed to the code synthesizer. The additional directives follow the format specified in Appendix A. To generate naïve code, the reuse directives are simply stripped off and the remaining clauses compile directly.

B.1 Append

```
insert( Element, [], [ Element ] ).

insert( Element, [ Head | Tail ], [ Element, Head | Tail ] ) :-
    Element < Head.

insert( Element, [ Head | Tail ], [ Head | NewTail ] ) :-
    Element >= Head,
    insert( Element, Tail, NewTail ).



:- pred_reuse( insert/3, [ 2 ], [ 3 ], [ entry, [ orphan ],
    [ ( 2, list ) ] ] ).

:- clause_reuse( insert/3/1, [ ( [ 2 ], [ 2, 3 ] ) ],
    [] ).

:- clause_reuse( insert/3/2, [ ( [ 1, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 1 ], [ 2, 3, 1 ] ),
        ( [ 1, 2, 1 ], [ 1, 2, 3, 1 ] ),
        ( [ 2, 2, 1 ], [ 2, 2, 3, 1 ] ) ],
    [] ).

:- clause_reuse( insert/3/3, [ ( [ 2, 1 ], [ 3, 1 ] ),
        ( [ 1, 2, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 2, 1 ], [ 2, 2, 2 ] ) ],
    [ ( [ 3, 2, 2 ], old( [ 2, 1 ], 2, 2 ) ) ] ).
```

B.2 Delete

```
delete( _, [], [] ).

delete( Old, [ Old | OldTail ], NewTail ) :-
    delete( Old, OldTail, NewTail ).

delete( Old, [ Other | OldTail ], [ Other | NewTail ] ) :-
    Old =Other,
    delete( Old, OldTail, NewTail ).



:- pred_reuse( delete/3, [ 2 ], [ 3 ], [ entry, [ orphan ],
    [ ( 2, list ) ],
    [ ( stray, list ) ] ] ).

:- clause_reuse( delete/3/1, [ ( [ 2 ], [ 3 ] ) ], [] ).

:- clause_reuse( delete/3/2, [ ( [ 1, 2, 1 ], [ 1, 2 ] ),
         ( [ 2, 2, 1 ], [ 2, 2 ] ) ],
    [ ( [ 3, 2 ], outside( 3 ) ) ] ).

:- clause_reuse( delete/3/3, [ ( [ 2, 1 ], [ 3, 1 ] ),
         ( [ 1, 2, 1 ], [ 1, 3, 1 ] ),
         ( [ 2, 2, 1 ], [ 2, 2, 2 ] ) ],
    [ ( [ 3, 2, 2 ], old( [ 2, 1 ], 2, 2 ) ) ] ).
```

B.3 Insert

```
insert( Element, [], [ Element ] ).

insert( Element, [ Head | Tail ], [ Element, Head | Tail ] ) :-
    Element < Head.

insert( Element, [ Head | Tail ], [ Head | NewTail ] ) :-
    Element >= Head,
    insert( Element, Tail, NewTail ).



:- pred_reuse( insert/3, [ 2 ], [ 3 ], [ entry, [ orphan ],
    [ ( 2, list ) ] ] ).

:- clause_reuse( insert/3/1, [ ( [ 2 ], [ 2, 3 ] ) ],
    [] ).

:- clause_reuse( insert/3/2, [ ( [ 1, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 1 ], [ 2, 3, 1 ] ),
        ( [ 1, 2, 1 ], [ 1, 2, 3, 1 ] ),
        ( [ 2, 2, 1 ], [ 2, 2, 3, 1 ] ) ],
    [] ).

:- clause_reuse( insert/3/3, [ ( [ 2, 1 ], [ 3, 1 ] ),
        ( [ 1, 2, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 2, 1 ], [ 2, 2, 2 ] ) ],
    [ ( [ 3, 2, 2 ], old( [ 2, 1 ], 2, 2 ) ) ] ).
```

B.4 Merge

```
merge( [ Head1 | Tail1 ], [ Head2 | Tail2 ], [ Head1 | NewTail ] ) :-
    Head1 =< Head2,
    merge( Tail1, [ Head2 | Tail2 ], NewTail ).

merge( [ Head1 | Tail1 ], [ Head2 | Tail2 ], [ Head2 | NewTail ] ) :-
    Head1 > Head2,
    merge( [ Head1 | Tail1 ], Tail2, NewTail ).

merge( [], [ Head | Tail ], [ Head | Tail ] ).

merge( List, [], List ).



:- pred_reuse( merge/3, [ 1, 2 ], [ 3 ], [ entry, [ orphan ],
    [ ( 1, list ) ],
    [ ( 2, list ) ] ] ).

:- clause_reuse( merge/3/1, [ ( [ 1, 1 ], [ 3, 1 ] ),
        ( [ 1, 1, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 1, 1 ], [ 1, 2, 2 ] ),
        ( [ 2, 1 ], [ 2, 2, 2 ] ),
        ( [ 1, 2, 1 ], [ 1, 2, 2, 2 ] ),
        ( [ 2, 2, 1 ], [ 2, 2, 2, 2 ] ) ],
    [ ( [ 3, 2, 2 ], old( [ 1, 1 ], 2, 1 ) ) ] ).

:- clause_reuse( merge/3/2, [ ( [ 1, 1 ], [ 1, 2, 2 ] ),
        ( [ 1, 1, 1 ], [ 1, 1, 2, 2 ] ),
        ( [ 2, 1, 1 ], [ 2, 1, 2, 2 ] ),
        ( [ 2, 1 ], [ 3, 1 ] ),
        ( [ 1, 2, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 2, 1 ], [ 2, 2, 2 ] ) ],
    [ ( [ 3, 2, 2 ], old( [ 2, 1 ], 2, 2 ) ) ] ).

:- clause_reuse( merge/3/3, [ ( [ 2 ], [ 3 ] ),
        ( [ 1, 2 ], [ 1, 3 ] ),
        ( [ 2, 2 ], [ 2, 3 ] ) ],
    [] ).

:- clause_reuse( merge/3/4, [ ( [ 1 ], [ 3 ] ) ], [] ).
```

B.5 Split

```
split( [], [], [] ).

split( [ Head ], [], [ Head ] ).

split( [ Head1, Head2 | Tail ], [ Head1 | Tail1 ], [ Head2 | Tail2 ] ) :-
    split( Tail, Tail1, Tail2 ).



:- pred_reuse( split/3, [ 1 ], [ 2, 3 ],
    [ entry, [ orphan, orphan ],
    [ ( stray, list ), ( 1, list ) ] ] ).


:- clause_reuse( split/3/1, [ ( [ 1 ], [ 2 ] ) ], [] ).

:- clause_reuse( split/3/2, [ ( [ 1 ], [ 3 ] ),
        ( [ 1, 1 ], [ 1, 3 ] ),
        ( [ 2, 1 ], [ 2, 3 ] ) ],
    [] ).

:- clause_reuse( split/3/3, [ ( [ 1, 1 ], [ 2, 1 ] ),
        ( [ 2, 1, 1 ], [ 3, 1 ] ),
        ( [ 1, 1, 1 ], [ 1, 2, 1 ] ),
        ( [ 1, 2, 1, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 2, 1, 1 ], [ 1, 2 ] ) ],
    [ ( [ 2, 2 ], oldStray( [ 1, 1 ], 2 ) ),
        ( [ 3, 2 ], old( [ 2, 1, 1 ], 2, 1 ) ) ] ).
```

B.6 Take

```
take_first([],[],[]).

take_first( [ [ Head | Tail ] | Lists ], [ Head | Heads ], [ Tail | Tails ] ) :-
    take_first( Lists, Heads, Tails ).



:- pred_reuse( take_first/3, [ 1 ], [ 2, 3 ],
    [ entry,
    [ orphan, orphan ],
    [ ( stray, list ), ( 1, list ) ] ] ).


:- clause_reuse( take_first/3/1, [ ( [ 1 ], [ 3 ] ) ], [] ).

:- clause_reuse( take_first/3/2, [ ( [ 1, 1 ], [ 3, 1 ] ),
        ( [ 1, 1, 1 ], [ 2, 1 ] ),
        ( [ 1, 1, 1, 1 ], [ 1, 2, 1 ] ),
        ( [ 2, 1, 1, 1 ], [ 1, 3, 1 ] ) ],
    [ ( [ 2, 2 ], oldStray( [ 1, 1, 1 ], 2 ) ),
        ( [ 3, 2 ], old( [ 1, 1 ], 2, 1 ) ) ] ).
```

B.7 Tree

```
tree_insert( Elem, tree( Node, Left, Right ), tree( Node, NewLeft, Right ) ) :-
    Elem < Node,
    tree_insert( Elem, Left, NewLeft ).

tree_insert( Elem, tree( Node, Left, Right ), tree( Node, Left, NewRight ) ) :-
    Elem >= Node,
    tree_insert( Elem, Right, NewRight ).

tree_insert( Elem, leaf, tree( Elem, leaf, leaf ) ).


:- pred_reuse( tree_insert/3, [ 1, 2 ], [ 3 ], [ entry, [ orphan ],
    [ ( 2, term ) ] ] ).


:- clause_reuse( tree_insert/3/1, [ ( [ 1, 1 ], [ 1, 2, 2 ] ),
        ( [ 2, 1 ], [ 3, 1 ] ),
        ( [ 1, 2, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 2, 1 ], [ 2, 2, 2 ] ),
        ( [ 3, 2, 1 ], [ 3, 3, 1 ] ) ],
    [ ( [ 3, 2, 2 ], old( [ 2, 1 ], 2, 2 ) ) ] ).

:- clause_reuse( tree_insert/3/2, [ ( [ 1, 1 ], [ 1, 2, 2 ] ),
        ( [ 2, 1 ], [ 3, 1 ] ),
        ( [ 1, 2, 1 ], [ 1, 3, 1 ] ),
        ( [ 2, 2, 1 ], [ 2, 3, 1 ] ),
        ( [ 3, 2, 1 ], [ 2, 2, 2 ] ) ],
    [ ( [ 3, 2, 2 ], old( [ 2, 1 ], 3, 2 ) ) ] ).

:- clause_reuse( tree_insert/3/3, [ ( [ 1 ], [ 1, 3 ] ),
        ( [ 2 ], [ 2, 3 ] ) ],
    [] ).
```

References

[1] B. Burton, G. Gudjonsson, and W. Winsborough, "An algorithm for computing alternating closure," Technical report CS-92-15, Pennsylvania State University Department of Computer Science, June 1992.

[2] I. Foster and W. Winsborough, "Copy avoidance through compile-time analysis and local reuse," in *Proceedings of the 1991 International Logic Programming Symposium*, MIT Press, 1991.

[3] A. Mulkers, W. Winsborough, and M. Bruynooghe, "Analysis of shared data structures for compile-time garbage collection in logic programs," in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press, 1990.

[4] R. Sundararajan, A. V. S. Sastry, and E. Tick, "Variable threadedness analysis for concurrent logic programs," in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1992.

[5] M. Codish, D. Dams, and E. Yardeni, "Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis," in *Proceedings of the Eighth International Conference on Logic Programming* (K. Furukawa, ed.), MIT Press, 1991.

[6] S. K. Debray, "On copy avoidance in single assignment languages," in *Tenth International Conference on Logic Programming*, MIT Press, 1993.

[7] S. Duvvuru, R. Sundararajan, E. Tick, A. Sastry, L. Hansen, and X. Zhong, "A compile-time memory-reuse scheme for concurrent logic programs," in *International Workshop on Memory Management*, (LNCS 637), Springer-Verlag, 1992.

[8] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe, "The impact of abstract interpretation: An experiment in code generation," in *Proceedings of the Sixth International Conference on Logic Programming* (G. Levi and M. Martelli, eds.), MIT Press, 1989.

[9] G. Gudjonsson and W. Winsborough, "Update in place: Overview of the Siva project." Submitted for conference publication, 1993.

[10] G. Gudjonsson, "Transforming prolog predicates to perform update in place." Unpublished technical notes, 1993.

[11] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, Kista, Sweden, March 1991.

[12] R. C. Haygood, *Aquarius Prolog User Manual*. Advanced Computer Architecture Laboratory, Los Angeles, September 1992.

[13] J. Schimpf, "setarg/3 in Sepia and ECLiPSe." Personal communication, April 1993.

[14] P. L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming*. PhD thesis, University of California, Berkeley, December 1990.

[15] Sun Microsystems, Inc., *Programming Utilities for the Sun Workstation*, 1985.

[16] R. B. Groves, "Transforming prolog predicates to perform update in place," Master's thesis, The Pennsylvania State University Department of Computer Science, May 1992.

[17] I. W. G. 17, "Programming language prolog–committee draft standard." ISO work item JTC1.22.22.1, April 1993.