A Simple Code Improvement Scheme for Prolog†

Saumya K. Debray

Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

Abstract

The generation of efficient code for Prolog programs requires sophisticated code transformation and optimization systems. Much of the recent work in this area has focussed on high level transformations, typically at the source level. Unfortunately, such high level transformations suffer from the deficiency of being unable to address low level implementational details. This paper presents a simple code improvement scheme that can be used for a variety of low level optimizations. Applications of this scheme are illustrated using low level optimizations that reduce tag manipulation, dereferencing, trail testing, environment allocation, and redundant bounds checks. The transformation scheme serves as a unified framework for reasoning about a variety of low level optimizations that have, to date, been dealt with in a more or less ad hoc manner.

---

† A preliminary version of this paper appeared in *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, June 1989. This work was supported in part by the National Science Foundation under grant number CCR-8702939.

1. Introduction

The generation of efficient code for Prolog programs requires sophisticated code transformation and optimization systems. Most of the recent work in this area has concentrated on high level transformations, typically at the source level [12, 19, 21, 22]. Such high level transformations have the advantage of being relatively simple to formulate and prove correct. However, they suffer from the deficiency that low level implementational details are often simply not expressible at the source level. As a result, after all appplicable high level transformations have been carried out, the programmer still finds himself penalized by low level inefficiencies that he is unable to overcome.

This paper presents a simple code transformation scheme that can be used for a variety of low level optimizations. It serves also as a unified framework for describing and reasoning about a variety of different low level optimizations that have, to date, been dealt with in a more or less ad hoc manner. Like the optimizations described by Mellish [17] and Mariën et al. [13], these are at the level of intermediate code, or virtual machine, instructions; they are somewhat higher level than the machine code level optimizations described by Turk [26].

The transformation scheme consists of a *hoisting* transformation on flow graphs, together with three generic transformations on basic blocks: *code introduction*, *code elimination*, and *code replacement*. Code hoisting is a transformation that is generally applicable; specific optimization algorithms are obtained by specifying particular instruction sequences that may be introduced at or eliminated from a point, or mutated in specific ways within a basic block, together with conditions under which this may be done. These transformations may be augmented by two transformations on flow graphs, called *node splitting* and *edge splitting*, that are applicable to all flow graphs, and always preserve program behavior. Because optimization-specific aspects of transformations are usually local to basic blocks, implementation and verification of optimizations is simplified. Applications of this scheme are illustrated using five low level optimizations: reduction of redundant tag manipulation operations, dereferencing, trail testing, environment allocation and bounds checking. These techniques may also be applicable to other low level optimizations, e.g. those of Mariën et al. [13], and the elimination of some redundant stack and heap overflow tests.

In work related to this, low level optimization of Prolog programs has been considered by, among others, Mariën et al. [13], Mellish [17], Meier [16], and Van Roy et al. [27]. Turk discusses a number of optimizations at the machine-code level, including the delaying of environment allocation [26]. The reduction of redundant dereferencing and trailing via global dataflow analysis has been addressed by a number of researchers [10, 13, 23, 24, 28]. Meier considers a number of optimizations, such as that of environment reuse (discussed in Section 4) in the context of loop optimization of Prolog programs [15].

The reader is assumed to be acquainted with the basic terminology of logic programming. The examples in the paper are based on a virtual machine model that resembles the Warren Abstract Machine [30] in many ways, especially in the parameter passing mechanism; however, we will take some liberties with the instruction set, choosing instructions to illustrate specific aspects of a particular optimization. It should be emphasized that the transformation scheme is not dependent on the WAM in any way, and applies equally to other machine models. Indeed, the transformations are not restricted to Prolog, and can be extended to other control strategies by appropriately defining the notions of ‘‘basic block’’ and ‘‘flow graph’’.

It is assumed that the predicates under consideration are *static*, i.e. any code for that predicate that can be executed at runtime is available for inspection by the compiler. This precludes predicates that can be modified at runtime via *assert* or *retract*, and predicates that contain dynamic goals of the form *call*(X) where X is a variable. In this context, it should be noted that the code transformations discussed may require information about the program, e.g. the type of a variable or the contents of a register at a program point; this may require dataflow analysis, which has to take primitives like *assert*/1 and *call*/1 into account, and may impose restrictions on their use [7].

2. Preliminaries

2.1. Basic Blocks and Flow Graphs in Prolog

The notions of ‘‘basic block’’ and ‘‘flow graph’’ are well known in traditional compiler theory. A *basic block* is a sequence of (intermediate code) instructions with a single entry point and single exit point: execution of a basic block can start only at its entry point; and control can leave a basic block only at its exit point. Thus, if control enters a basic block, each instruction in that block will be executed. A *flow graph* for a procedure is a directed graph whose nodes are the basic blocks of that procedure, where there is an edge from a node B_1 to a node B_2 if it is possible for control to enter B_2 immediately after it has left B_1. If there is an edge from B_1 to B_2 in a flow graph, then B_1 is said to be a *predecessor* of B_2, and B_2 is said to be a *successor* of B_1.

When dealing with logic programs, this definition of a basic block does not work quite as desired, because whereas most operations in traditional languages have only a single continuation (the ‘‘success continuation’’, which is usually the next instruction), operations in logic programming languages, e.g. at the WAM code level, typically have two continuations: the ‘‘success continuation’’ and the ‘‘failure continuation’’. As a result, dividing the WAM code for a Prolog program into basic blocks using the traditional definition typically results in a large number of trivial basic blocks, each containing a single WAM instruction. The resulting flow graph is large and messy, with much of the control flow structure of the original program obscured, and is not very amenable to compile-time optimization.

It is therefore necessary to change the notion of a ‘‘basic block’’ slightly for logic programs. We propose the following definition:

Definition: A *basic block* in a logic program is a maximal sequence of instructions **I** with the following properties:

(i) **I** has a single entry point, i.e. execution can enter **I** only through its first instruction; and (*ii*) **I** has a single successful exit point: if control enters **I** and each instruction in **I** succeeds, then each instruction in **I** is executed exactly once. ♦

Note that instructions corresponding to procedure calls, e.g. the **call** instruction of the WAM, need not terminate a basic block, since a successful return from a procedure call corresponds to the successful execution of the **call** instruction, and this is treated like the successful execution of any other instruction.

With this definition of basic blocks, execution can leave a basic block in two ways: via success, and by failure. If control flows from a block B_1 to a block B_2 via successful execution, the changes to variables and registers effected by B_1 are visible to B_2. However, if control flows from B_1 to B_2 via failure, then changes to the virtual machine state effected by B_1 will in general be invisible to B_2. This can be made explicit by using two kinds of edges in the flow graph: *success edges* and *failure edges*:

Definition: The flow graph of a clause is a directed graph whose nodes are the basic blocks of that clause. There is a success edge from a block B_1 to a block B_2 if control, upon leaving B_1 successfully, can go immediately to B_2. There is a failure edge from B_1 to B_2 if execution, on failing in B_1, can go immediately to B_2.

The flow graph for a predicate consists of the flow graphs for each of its clauses, together with an *entry node* that is distinct from the nodes in the flow graphs for the clauses. The edges of this flow graph are defined as follows:

(1) there is a success edge from the entry node to the header node of the flow graph of the first clause of the predicate;

(2) there is a failure edge from the header of the flow graph for a clause C_i to the header of the flow graph for a clause C_j if execution backtracks to C_j when C_i fails.

♦

Execution always enters the flow graph of a predicate through its entry node. Information flow during the execution of a predicate can be made more explicit by elaborating slightly on its flow graph: the result is a graph called its augmented flow graph.

Definition: An *augmented flow graph* for a predicate *p* is a directed graph *G* whose nodes are those for its flow graph, and whose edges are defined as follows:

(1) if there is a success (failure) edge from B_1 to B_2 in the flow graph for *p*, then there is a success (failure) edge from B_1 to B_2 in *G*;

(2) if there is a success edge from B_1 to B_2 and a failure edge from B_2 to B_3 in *G*, where $B_1 \neq B_3$, such that execution can succeed through B_1 into B_2 and then fail back into B_3, then there is a success edge from B_1 to B_3 in *G*. ♦

The reasoning behind the edges added in (2) is as follows: if execution can go successfully from B_1 to B_2 and then fail into B_3, then B_3 will in general see changes to the machine state effected by B_1 but not those effected by B_2. From the point of view of B_3, therefore, it is as if execution had succeeded through B_1 and gone immediately into B_3. This justifies adding a success edge from B_1 to B_3 in the augmented flow graph. The discussion that follows will generally concern itself only with augmented flow graphs, and hence not explicitly distinguish between ''flow graphs'' and ''augmented flow graphs''. Depending on the implementation, some changes to machine registers or flags effected by B_2 may be visible to B_3: these can be taken into account during the transformation using the failure edge from B_2 to B_3. An example of an augmented flow graph is given in Figure 1.

As mentioned above, the difference between control flow along a success edge and that along a failure edge is that certain variables may have their values reset when a failure edge is traversed. This can be made explicit by associating a set of variables *undo*(e) with each failure edge e in a flow graph; this set is called the *undo set* of the edge e. If v is a variable in the undo set of an edge e leaving a node n in a flow graph, then any instantiation of v that takes place after entering the node n is undone when execution fails back from n along e.

2.2. Variable Liveness

The ''liveness'' of a variable is a familiar notion from traditional compiler theory: a variable is live at a point in a basic block if there is some execution path, from that point, along which the value of that variable may be used before it is (re)defined. The notion of variable liveness in our case is conceptually the same as this, with minor extensions to handle the difference between success and failure edges:

Definition: A variable v is *used* at a point p in a basic block if and only if v is an operand of the instruction at the point p. A variable v is *defined* at a point p in a basic block if and only if the execution of the instruction at p assigns a value to v.

Intuitively, a use of a variable corresponds to ''reading'' the value of that variable, while a definition of a variable corresponds to ''writing'' a value into it. Note that in some cases, operands of an instruction may be implicit, e.g. the `H` register in the `getlist` instruction in the WAM.

FIGURE 1

Definition: A variable v is *live* at a point p in a basic block B in a flow graph if and only if

(1) there is a point p' in B that is after the point p, such that v is used at p' and is not defined at any point between p and p'; or

(2) v is not defined in B after the point p, and there is a basic block B' in the flow graph such that there is a success edge from B to B', and v is live at the entry to B'; or

(3) v is not defined in B after the point p, and there is a basic block B' in the flow graph such that there is a failure edge e from B to B', v does not occur in *undo*(e), and v is live at the entry to B'. ♦

The notion of variable liveness finds numerous applications in code generation and optimization. An important application of liveness is in defining the correctness conditions for the code hoisting transformation described in the next section.

3. The Transformation Scheme

The transformation scheme consists of a pair of dual transformations on flow graphs called *code hoisting*, and three generic transformations on basic blocks: *code introduction*, *code elimination*, and *code replacement*. The hoisting transformations are generally applicable to flow graphs that satisfy certain criteria. Particular code optimization algorithms are obtained by giving specific code introduction, code elimination and code replacement sets, i.e. sets of instruction sequences, together with conditions under which an instruction sequence may be inserted at or deleted from a point within a basic block, or replaced by another instruction sequence. These may be augmented by two transformations on flow graphs, called *node splitting* and *edge splitting*, that are applicable to all flow graphs and always preserve program behavior.

3.1. Code Hoisting

Code hoisting consists of two dual transformations on flow graphs, called *upward* and *downward* code hoisting respectively.

3.1.1. Upward Code Hoisting

Upward code hoisting is defined as follows: let **A** and **B** be sets of basic blocks satisfying (i) for any $A \in$ **A**, if B is a successor of A then $B \in$ **B**; and (ii) for any $B \in$ **B**, if A is a predecessor of B then $A \in$ **A**. Let every block $B \in$ **B** start with a sequence of instructions S. Then, upward code hoisting deletes the instruction sequence S from the beginning of each block in **B**, and inserts it at the end of each block in **A** (if a block A in **A** ends in a transfer-of-control instruction I, then the sequence S is inserted immediately before I, as shown in Figure 2). If any of the blocks in **B**, say B_k, is the entry node of the flow graph, then a new block A is created containing only the instruction sequence S, A is made the new entry node of the flow graph (so that execution now enters the flow graph through A), and B_k is made the only successor of A.

FIGURE 2

To see when this transformation can be applied, consider a block B0 with two successors, B1 and B2. Let the instruction sequence in B0 before hoisting be T0 followed by a transfer-of-control instruction (which can be a conditional or unconditional jump, or an instruction that creates or manipulates a choice point, e.g. a *try*, *retry* or *trust* instruction). Let the instruction sequences in B1 and B2 be, respectively, S followed by T1 and S followed by T2. The relevant fragments of the flow graph before and after hoisting are shown in Figure 2.

First, observe that in the flow graph before hoisting, the instruction sequence S is executed after the branch instruction in B0; after hoisting, however, S is executed before the branch instruction. It is necessary to ensure, therefore, that S does not define anything used in the conditional jump. The transfer-of-control instruction at the end of B0 may itself define registers or variables used in S, e.g. if it is a *try*, *retry* or *trust* instruction. In general, therefore, it is necessary to ensure that hoisting does not disturb definition-use relationships between the blocks involved in the transformation.

While this condition ensures correct forward execution behavior, it does not guarantee proper execution upon backtracking. To see this, suppose that execution backtracks to B2 upon failure in B1. In the flow graph before hoisting, the instruction sequence executed, when B2 is entered upon backtracking from B1, is S followed by T2; however, after code hoisting, only T2 is executed when execution backtracks into B2. If S defines any variable or register that is used in T2 but is not saved in the choice point for these execution paths and restored upon backtracking, then the transformation can result in incorrect execution. Also, if S has any externally visible side effects, e.g. through *assert*, *write*, etc., then the code before and after hoisting behave differently. To ensure correctness of backward execution, therefore, it is necessary to ensure that either (*i*) execution cannot backtrack from B1 to B2 (i.e. the two execution paths are mutually exclusive); or (*ii*) the instructions being hoisted do not have any externally visible side effects, and if they define any variable, register or flag that is live at a subsequent point in the block they are hoisted from, then the value that variable, register or flag is restored upon backtracking (strictly speaking, it is necessary to require restoration upon backtracking only if the forward execution through B1 and its successors can alter the values of such variables or registers). A strong mutual exclusion condition is needed here, since execution cannot be permitted to fail back into B2 once it has entered B1. Thus, in this case it must be possible to determine exactly which execution path to try simply from the instruction sequence T0 in B0.

3.1.2. Downward Code Hoisting

This is the dual of upward code hoisting. Let **A** and **B** be sets of basic blocks satisfying (i) for any $B \in$ **B**, if A is a predecessor of B then $A \in$ **A**; and (ii) for any $A \in$ **A**, if B is a successor of A then $B \in$ **B**. Let every block in **A** end with a sequence of instructions S. Then, downward code hoisting deletes S from the end of each block in **A**, and inserts S at the beginning of each block in **B**.

The correctness conditions for downward code hoisting are analogous to those for upward hoisting. Its utility lies mainly in the optimization to reduce the amount of redundant environment allocation, discussed in the next section, and in the reduction of redundant tag manipulation operations across procedure boundaries, discussed in [5].

3.2. Code Introduction

This transformation on basic blocks is specified by a set of pairs $\langle S, P\rangle$, called the *code introduction set*, where S is a sequence of instructions (or instruction schemas), and P is a condition. If $\langle S, P\rangle$ is in the code introduction set of an optimization, then at any point within a basic block where the condition P is satisfied, the instruction sequence S can be inserted without affecting the behavior of the program.

The primary purpose of code introduction is to allow code hoisting to be performed. This is illustrated in the applications discussed in the next section. The following points should be noted:

(1) The presence of a pair $\langle S, P\rangle$ in the code introduction set of an optimization means that whenever P is satisfied at a point within a basic block, S can be inserted at that point without affecting the behavior of the program. It is up to the designer of that optimization to ensure that this is indeed the case. However, because the transformation is local to a basic block, the task of verifying that program behavior is unaffected by the introduction of S at any point where P holds can often be carried out by local reasoning, thereby simplifying specification and reasoning about such low level transformations considerably.

(2) In general, the code introduction set of an optimization specifies only when code *can* be introduced at a program point without altering the behavior of the program, not when it *should* be introduced. However, pragmatic considerations, e.g. cost criteria based on which the compiler may decide whether code introduction is worth performing at a particular program point, may also be incorporated into the condition associated with each code fragment in the code introduction set if desired.

3.3. Code Elimination

This is specified by a set of pairs called the *code elimination set*, which consists of a set of pairs $\langle S, P\rangle$, where S is a sequence of instructions (or instruction schemas), and P is a condition. If $\langle S, P\rangle$ is in the code elimination set of an optimization, then at any point within a basic block where the instruction sequence S occurs and the condition P is satisfied, S can be eliminated without affecting the behavior of the program.

While code elimination is conceptually the dual of code introduction, their functions are very different: code introduction is intended to make code hoisting possible; this hoisting is then intended to make code elimination possible; finally, the actual optimization is achieved by code elimination. In general, therefore, the code introduction set and code elimination set of an optimization are different. As with code introduction, it is usually the case that only a few instruction sequences need be considered for any particular optimization.

3.4. Code Replacement

This is specified by a set of triples $\langle S_1, S_2, P\rangle$ called the *code replacement set*, where S_1 and S_2 are sequences of instructions or instruction schemas, and P is a condition. If $\langle S_1, S_2, P\rangle$ is in the code replacement set of an optimization, then at any point in a basic block where the instruction sequence S_1 is encountered and P is satisfied, S_1 can be replaced by S_2. The following points are worth mentioning in this context:

(1) The *Code Introduction* and *Code Elimination* transformations can be seen as special cases of Code Replacement: in the former, the code replacement set is of the form $\langle \varepsilon, S, P\rangle$, where ε denotes the empty sequence, while in the latter the code replacement set is of the form $\langle S, \varepsilon, P\rangle$. Strictly speaking, therefore, a single optimization-specific local transformation – namely, Code Replacement – suffices for our purposes. However, Code Introduction and Code Elimination play very specific roles in our transformation scheme, and also make the transformations easier to understand conceptually, so we present them separately as distinct transformations even though this is technically not necessary.

(2) In most of the optimizations considered in this paper, Code Replacement involves moving a sequence of instructions – very often, just a single instruction – from one point in a basic block to another. Verification of the correctness of Code Replacement under such circumstances can usually be carried out by purely local reasoning, which is both simple and easy to implement. However, more general transformations can also be specified, as the following example illustrates: Consider the instruction sequence

move a0@, r1
move a1@, r2
r2 := r1 + r2
move r2, a1@

where *a1@* denotes an indirect reference through *a1*. If both *r1* and *r2* are dead at the end of this sequence, then this instruction sequence can be replaced by the single instruction

a1@ := a1@ + a2@.

3.5. Auxiliary Transformations

There may be situations where the code hoisting transformation described earlier cannot be carried out because of the structure of the flow graph. It may also happen that hoisting is possible, but practically undesirable, e.g. because it introduces code into a loop, as illustrated in Figure 3. In these cases, it is sometimes possible to transform the flow graph in a manner that makes it possible for the transformations described earlier to be applied. We consider two such transformations here: *node splitting* and *edge splitting*. These transformations are applicable to all flow graphs, and always preserve program behavior.

3.5.1. Node Splitting

Let B be a basic block in a (augmented) flow graph G, and let the predecessors and successors of B be denoted by $preds(B)$ and $succs(B)$ respectively. A k-way splitting of the node B, $k > 0$, is carried out as follows: let $\{preds_1(B), \ldots, preds_k(B)\}$ and $\{succs_1(B), \ldots, succs_k(B)\}$ be partitionings of the sets $preds(B)$ and $succs(B)$ respectively, each containing k elements, such that none of the elements $preds_i(B)$ or $succs_i(B)$ is empty. The node B in G is then replaced by k copies $B_1, \ldots, B_k$ of B, such that the predecessors of B_i are the nodes in partition $preds_i(B)$, while its successors are the nodes in partition $succs_i(B)$. If there was a failure edge from the block B to a block B' in the flow graph before splitting, then there is a failure edge from each of the blocks $B_1, \ldots, B_k$ to B' in the flow graph after splitting; if there was a failure edge from a block B'' to B in the flow graph before splitting, then there is a failure edge from B'' to each of the blocks $B_1, \ldots, B_k$ in the flow graph after splitting.

3.5.2. Edge Splitting

Let B_1 and B_2 be two basic blocks in a flow graph G, such that there is a success edge e from B_1 to B_2. *Edge splitting* refers to splitting e by inserting an empty basic block B, i.e., one containing no instructions, between B_1 and B_2. In other words, a new node B – consisting of an empty basic block – is introduced into the flow graph, and the edge e from B_1 to B_2 is replaced by an edge from B_1 to B and one from B to B_2.

This transformation offers another solution to the situation illustrated in Figure 3: the edge from the basic block in the loop to the block containing the instruction sequences S and I2 can be split using an empty block B. If the instruction sequence S is now hoisted, it is introduced into B but not into the loop.

4. Applications to Code Optimization

This section describes a number of applications of the transformation scheme to low level code optimization. The essential idea, in each case, is to repeatedly perform code hoisting and code elimination; to

---

FIGURE 3

---

make hoisting possible, code replacement and code introduction are carried out where necessary.

4.1. Tag Manipulation Reduction

Objects that are passed around in Prolog implementations are typically associated with bit patterns, called *tags*, that indicate their types. Runtime operations often follow the pattern: (*i*) examine the tag bits of the operands to ensure that they are of the appropriate type(s); (*ii*) untag each operand; (*iii*) perform the operation; and (*iv*) tag the result. While one or more of these steps can be omitted for some operations by careful choice of the tagging scheme, some tag manipulation is necessary in general, and can, in many cases, lead to redundant tagging/untagging and type checking that can incur a significant penalty. As an example, consider the following program to sum the values in a list:

sumlist(*L*, *N*) :– *sumlist*(*L*, 0, *N*).

sumlist([], *N*, *N*).
sumlist([*H*|*L*], *K*, *N*) :– *K1* is *K* + *H*, *sumlist*(*L*, *K1*, *N*).

Consider a sequence of activations of the recursive clause in a call to *sumlist*/3: first, the expression '*K* + *H*' is evaluated: this involves checking that the variables *H* and *K* are instantiated to numbers, untagging each of them, adding them together, then tagging the result and unifying the tagged result with the variable *K1*. Immediately after this, in the next recursive call, however, the tagged sum from the previous step is again checked for numberhood and untagged, after which it participates in an addition, and the result is again tagged. This is repeated at each invocation of the recursive call. Clearly, this is wasteful: instead, the second argument to *sumlist*/3 can be untagged once, at the point of entry, and carried around the loop untagged. (To simplify the discussion, we are assuming that the only numeric objects are integers. These ideas can be extended to deal with floating point values by having two copies of the code, one optimized for the (expected) case of integers, the other representing the ''general case''.)

Actually, some care is necessary if untagged objects are to be passed around. In the example above, notice that when the recursion terminates, the second and third arguments of *sumlist*/3 are unified. Since unification generally needs to know the types of its operands, it is necessary to restore the tag on the second argument of *sumlist*/3 before this unification is carried out. In general, other operations, such as indexing, may also require tagged operands. The compiler therefore has to ensure that, while untagged objects are passed around and manipulated wherever possible, tags are correctly restored where necessary. Moreover, garbage collection and debugging in the presence of untagged objects require additional support in order to correctly identify untagged objects [5].

Two instructions are assumed for explicit tag manipulation. The instruction "**untag** *u*, *t*" checks that the object *u* has the tag *t*: if so, it removes the tag, converting *u* to its untagged form; otherwise, it fails. The instruction "**tag** *u*, *t*" adds the tag *t* to the object *u*, i.e. converts *u* to its *t*-tagged form. The transformation is defined by the following:

Code Introduction: Suppose it is known, at a program point, that an object *u* is of type *t* (this information must be obtained separately, e.g. via dataflow analysis), then the instruction sequence

untag *u*, *t*

tag *u*, *t*

may be introduced at that program point.

Code Elimination: If the sequence of instructions

tag *u*, *t*

untag *u*, *t*

occurs at any point in a program, then it may be deleted.

Code Replacement: This involves moving **untag** instructions to the tops of their basic blocks, and **tag** instructions to the bottoms of their blocks. If an instruction "**untag** *u*, *t*" is being migrated across a sequence of instructions *I* in this process, then it is necessary to ensure that *I* does not contain any procedure calls, and does not define or use *u*; a similar comment applies to the movement of **tag** instructions.

The transformation can be illustrated by considering the *sumlist*/3 predicate above. A flow graph for this predicate before the transformation is carried out is given in Figure 4(a). The transformation proceeds as follows:

1. *Code Replacement*:

   The "**untag** *r2*, int" instruction in block B2 is migrated to the top of that basic block.

2. *Code Introduction*:

   The instruction sequence "**untag** *r2*, int; **tag** *r2*, int" is introduced at the top of basic block B1. Here, we assume that dataflow analysis has already established that the second argument to *sumlist*/3 is always an integer.

3. *Code Hoisting*:

   The instruction "**untag** *r2*, int" is hoisted from blocks B1 and B2 into B0.

4. *Code Hoisting*:

   The instruction "**untag** *r2*, int" is hoisted again, this time from block B0. Since B0 is the entry node of the procedure, a new entry node is created containing just this instruction. This step also results in the hoisted instruction being inserted at the bottom of block B2, since B2 is a predecessor of B0.

FIGURE 4

---

5. *Code Elimination*:

The sequence of instructions "**tag** *r2,* int; **untag** *r2,* int" at the end of block B2 is eliminated.

The resulting flow graph, shown in Figure 4(b), achieves exactly what was intended: the second argument to *sumlist*/3 is untagged once at the entry to the loop, and then passed around the loop untagged. This avoids the cost of repeated removal and restoration of tags during iteration. However, at the end of the loop, just before an instruction that demands that it be tagged, its tag is restored.

An important point to note is that all aliases of an object should be known to have the same representation, tagged or untagged, at any particular point in a program. This is true even though the transformations described here are not applied globally to a program, but rather only to the flow graph for a procedure, because if two different variables within a procedure can be aliases at some point, i.e., can dereference to the same location, then the compiler must ensure that the code generated to access this location is consistent with its representation no matter which of the aliased variables is used to access it. Aliases may be determined using dataflow analyses designed for this purpose (e.g. see [6, 8, 9, 18]). Alternatively, since registers cannot have pointers to them, the optimization may be limited to objects resident in registers – the latter alternative, while simpler, is conservative and may fail to exploit the optimization to its fullest.

If untagged objects are passed around at runtime, it is necessary to ensure that they are not misinterpreted, e.g. during garbage collection or debugging. This can be done by storing additional information about the representation of local objects and registers in the symbol table entry for each predicate. The idea is to augment the symbol table entry of each procedure with a list that specifies ranges within the code for that procedure that contain untagged objects; and for each such range, the variables and registers that contain untagged values, together with the actual tags for each such variable and register. Since only one copy of this information is maintained, the space overhead is not very large. The debugger or garbage collector uses the value of the program counter to search this list for the appropriate address range, after which the information in the symbol table can be used to correctly identify all untagged objects in that procedure's environment. The scheme, which is similar in many ways to a proposal by Appel for strongly typed polymorphic languages like ML [2], is discussed in more detail in [5].

4.2. Dereferencing Reduction

One of the commonest operations in Prolog implementations is that of dereferencing. Because of this, repeated dereferencing of a variable or register can lead to a reduction in execution speed, because apart from the execution of redundant instructions, the conditional branch within a dereference loop can affect instruction locality, especially with regard to pipelining. Moreover, the repetition of dereferencing code can lead to an undesirable increase in the code size for a program, which can also adversely affect instruction cache and paging behavior.

As an example of redundant dereferencing, consider the *factorial* program:

fact(*N*, *F*) :– *fact*(*N*, 1, *F*).

$fact(0, F, F)$.

$fact(N, K, F) :- N > 0, K1$ is $N * K$, $N1$ is $N - 1$, $fact(N1, K1, F)$.

In a typical implementation, each clause dereferences the arguments separately, without regard to what other clauses may be doing; in many cases, moreover, variables may be dereferenced even though they have been dereferenced earlier and further dereferencing is unnecessary. Thus, the first clause for *fact*/3 will dereference its first argument and attempt to unify this value with 0. When this fails, execution backtracks to the second clause, where N is dereferenced once for the test '$N > 0$', once to evaluate the expression '$N - 1$', and once to compute the value of '$N * K$': a total of four dereference operations at each call, where one suffices. This happens even in (current versions of) sophisticated implementations such as Sicstus and Quintus Prolog [4, 11].

We assume that dereferencing is made explicit via an instruction "x := **deref**(y)" that dereferences the variable or register y and puts the result in x. The transformation is defined as follows:

Code Introduction: Let **instr**(... x, y ...) be any instruction that always dereferences an operand x and puts the dereferenced value in y. Then, if the instruction **instr**(... x, y ...) appears at a point in a program, the instruction "y := **deref**(x)" can be introduced immediately before it.

Code Elimination: If, at a given program point, it can be guaranteed that a variable or register x contains the result of ''fully dereferencing'' a variable or register y, and the instruction "x := **deref**(y)" appears at that point, then this instruction can be deleted.

Code Replacement: This consists of moving **deref** instructions to the tops of their basic blocks: Let I be a sequence of instructions that does not define y and does not define or use x, such that either (i) I does not contain any procedure calls, or (ii) y is guaranteed to be bound to a nonvariable term at the point immediately before I. Then, given the sequence of instructions

I
x := **deref**(y)

at a program point, the **deref** instruction may be migrated above I to yield

x := **deref**(y)
I

The reason an instruction "x := **deref**(y)" can be migrated across a procedure call if y is bound to a non-variable term is that unifications in the procedure being called will have no effect on what y dereferences to. On the other hand, if y dereferences to a variable X, unifications in the procedure call may result in a lengthening of the chain of pointers from X, because X may become bound to a pointer chain of nonzero length in the called procedure. In this case, dereferencing y before the procedure call may produce results that differ from those obtained by dereferencing it after the procedure call. Note that if x is a register and the instruction sequence I contains procedure calls, then the requirement that I should not define or use x extends to every execution path that can result from such calls. In practice, therefore, it may be simplest to not apply this transformation if x is a register and I contains procedure calls.

The transformation proceeds as before, by repeatedly performing code replacement, code hoisting, and code elimination. This can be illustrated by considering the *fact* program above. The initial flow graph is given in Figure 5(a). The transformation proceeds as before:

1. *Code Replacement*:

   The instruction "*r2* := **deref(***r2*)" in blocks B1 and B2 are migrated to the tops of their basic blocks. The "*r1* := **deref(***r1*)" instructions in block B2 are migrated to the top of B2.

2. *Code Elimination*:

   Of the three "*r1* := **deref(***r1*)" instructions at the beginning of block B1, two are eliminated because after the first such instruction, it can be guaranteed that *r1* contains the result of fully dereferencing *r1*.

3. *Code Hoisting*:

   The instruction sequence "*r1* := **deref**(*r1*); *r2* := **deref**(*r2*)" is hoisted from the tops of blocks B1 and B2 into block B0.

4. *Code Hoisting*:

   The instruction sequence "*r1* := **deref**(*r1*); *r2* := **deref**(*r2*)" is hoisted again, this time from block B0. Since B0 is the entry node for the procedure, a new entry node is created containing just these two instructions. This hoisting step also results in this instruction sequence being inserted at the bottom of block B2.

5. *Code Elimination*:

   The instruction "*r1* := **deref**(*r1*)" at the end of block B2 is eliminated, since properties of arithmetic operations can be used to guarantee that after the instruction "*r1* := *r1* − 1", the value in *r1* is fully dereferenced.

   The instruction "*r2* := **deref**(*r2*)" at the end of block B2 is eliminated, since properties of arithmetic operations guarantee that the value in *r2* is fully dereferenced after the instruction "*r2* := *r1* * *r2*", and *r2* is not redefined between this arithmetic instruction and the instruction dereferencing it.

In the resulting flow graph, illustrated in Figure 5(b), registers *r1* and *r2* are each dereferenced just once, at the entry to the procedure. In this example, the original code performs $5n + 3$ dereferencings to

---

FIGURE 5

---

compute *fact*(n) – five dereferences each time around the loop, and an additional 3 dereferences when the recursion terminates – while the optimized code requires only 3 dereferences altogether.

4.3. Redundant Trail Test Reduction

When a variable gets a binding during unification, it is generally necessary to determine whether or not it should be ‘‘trailed’’, i.e., have its address logged so that the binding may be undone on backtracking. A variable getting a binding (which may be either another variable, or a non-variable term) must be trailed if it is older than the most recent choice point. It is often the case, however, that trail tests at some program points are redundant, in the sense that the variable being tested is guaranteed to be younger (or guaranteed to be older) than the most recent choice point when execution reaches that point. This section describes the application of our code improvement scheme to the detection and elimination of some of the redundant trail tests that may occur in a program.

As an example of redundant trail testing, consider the following predicate, which removes duplicate elements from a list:

rem_dups([], []).
rem_dups([*H*|*L1*], *L2*) :– (*member*(*H*, *L1*) –> *L2* = *L3* ; *L2* = [*H*|*L3*]), *rem_dups*(*L1*, *L3*).

Assume that it is known that *rem_dups*/2 is always invoked with its first argument bound and the second argument free. Many implementations of Prolog will recognize that, in the unification ‘*L2* = *L3*’ in the body of the recursive clause for *rem_dups*/2, *L3* is a new variable that is necessarily younger than the most recent choice point, so that no trail test is necessary here; however, most current Prolog implementations will test whether *L2* needs to be trailed at the unification ‘*L2* = [*H*|*L3*]’. However, it is not difficult to see that no choice point is created during head unification in *rem_dups*/2, and any choice points that may have been created by the call to *member*/2 are discarded immediately upon return by the –> operator. Thus, the most recent choice point when execution reaches the unifications ‘*L2* = *L3*’ and ‘*L2* = [*H*|*L3*]’ is always the same as that at the entry to *rem_dups*/2, irrespective of the number of iterations the predicate has performed. It follows from this that it suffices to trail *L2* at most once, at the entry to the predicate, rather than once at each iteration in which the call to *member*/2 fails.

The instructions assumed to implement the *cut* and –> constructs of Prolog are "**save_cp** *u*" and "**cut_to** *u*": "**save_cp** *u*" stores a pointer to the most recent choice point in *u* (which may be a variable or a register), while "**cut_to** *u*" sets the most recent choice point to be that pointed at by *u*. Schemes similar to this are used to implement *cut* in many Prolog implementations, e.g. see [3]. Note that for any instruction sequence *I* that does not define *u*, the most recent choice point after the execution of the instruction sequence

save_cp *u*
I
cut_to *u*

is the same as that immediately before the execution of this sequence, irrespective of whether *I* succeeds or fails. The instruction "**trail** *u*" is used to make trail tests explicit: the instruction tests whether *u* dereferences to a variable that is older than the most recent choice point, and if so, pushes a pointer to this variable on the trail. Finally, the assertion **no_trail**(u) is true at a given program point if and only if it is not necessary to dereference and trail u if a binding is created for u at that point, i.e., if and only if either (i) u is guaranteed to be younger than the most recent choice point, or (ii) the location that u dereferences to is guaranteed to have been trailed since the most recent choice point was created. The following rules of inference guide the manipulation of **no_trail**(...) assertions:

NT1: **no_trail**(u) is true at the point immediately after an instruction "**trail** u". The justification for this is that u has already been trailed at this point, if necessary, so there is no need to trail it again right away.

NT2: Let "u := **newvar**(...)" be any instruction that binds u to a new variable (e.g., the **put_variable** instruction in the WAM), then **no_trail**(u) is true at the point immediately after such an instruction. The justification for this is that the variable that u is bound to immediately after such an instruction is guaranteed to be younger than the most recent choice point.

NT3: Let v be a variable that is guaranteed to be younger than the most recent choice point, and let **no_trail**(u) be true immediately before an instruction "v := u", then **no_trail**(v) is true immediately after this instruction. The justification for this is that immediately after this instruction is executed, v dereferences to the same location that u dereferences to. If u does not need to be trailed at that point, then the only possible reason for trailing v would be to reset, on backtracking, the binding created by this instruction. But since v is guaranteed to be younger than the most recent choice point, it is not necessary to explicitly reset its binding on backtracking. It follows that v does not need to be trailed at the point immediately after this instruction.

Let **no_trail**(u) be true immediately before an instruction "r := u", where r is a register, then **no_trail**(r) is true immediately after this instruction. The justification for this is similar to that above, under the assumption that the contents of a general purpose register are not, in general, restored on backtracking (unless, of course, it was saved in a choice point, which is not what we are considering here).

NT4: Let I be an instruction sequence satisfying: (i) I does not define u; and (ii) any choice points created during the execution of I are guaranteed to have been discarded by the time execution reaches the end of I. I may span basic block boundaries, provided that execution cannot branch into the middle of I. Then, if **no_trail**(u) is true immediately before I, then it is true immediately after I. The justification for this is follows from the fact that the most recent choice point when execution reaches the end of I is no younger than that at the beginning of I (but may be older).

NT5: If **no_trail**(u) is true at the end of every predecessor of a basic block B, then it is also true at the beginning of B.

The transformation is defined as follows:

Code Introduction: If u is guaranteed to be an uninstantiated variable at a given program point, then the instruction "**trail** u" may be introduced at that point. The justification behind this is that the only effect of the newly introduced "**trail** *u*" would be, on backtracking, to ‘‘unbind’’ to the location that *u* dereferences to: but since *u* is an unbound variable at that point anyway, this will not change the behavior of the program.

Code Elimination: If an instruction "**trail** *u*" occurs at a program point, and **no_trail**(*u*) is true at that point, then the "**trail** *u*" instruction may be deleted.

Code Replacement: This transformation is not used.

The transformation can be illustrated by considering the *rem_dups*/2 predicate given earlier. The flow graph for this predicate before transformation is shown in Figure 6(a). Assume that it is known, from mode information obtained either from user declarations or via global flow analysis, that the second argument to *rem_dups*/2 is always an uninstantiated variable. The transformation proceeds as follows:

1. *Code Introduction*:

   From the mode information assumed, it can be guaranteed that the second argument of *rem_dups*/2 is uninstantiated at every call to this predicate, so the instruction "**trail** *r2*" is introduced at the entry to the predicate, i.e., at the top of block B0.

   It can then be inferred, using NT1, that **no_trail**(*r2*) is true at the point immediately after the newly introduced "**trail** *r2*" instruction in block B0.

2. *Code Hoisting*:

   The instruction "**trail** *r2*" is hoisted from block B0. Since B0 is the entry node for the flow graph, this results in the creation of a new entry node B, containing only the instruction "**trail** *r2*", whose only successor is B0. This step results in the introduction of the instruction "**trail** *r2*" at the bottoms of blocks B3 and B4.

3. *Propagating* **no_trail**(...) *Assertions*:

(*a*) Since **no_trail**(*r2*) is true at the end of block B0, it follows from rule NT5 that **no_trail**(*r2*) is true at the beginning of basic blocks B1 and B2.

4. *Code Elimination*:

   Since **no_trail**(*r2*) is true at the beginning of block B1, the instruction "**trail** *r2*" at the beginning of this block can be deleted.

FIGURE 6

---

5. *Propagating* **no_trail**(...) *Assertions*:

Since **no_trail**(*r2*) is true at the beginning of block B2, it follows from rule NT4 that it is true immediately before the instruction "*Ys* := *r2*" in this block. From this, since *Ys* is a new variable that must be younger than the most recent choice point, it can be inferred using rule NT3 that **no_trail**(*Ys*) is true immediately after the instruction "*Ys* := *r2*" in this block.

6. *Propagating* **no_trail**(...) *Assertions*:

Using rule NT4, it can be inferred that **no_trail**(*Ys*) is true at the beginning of block B3, whence another application of NT4 shows that it is true immediately before the instruction "**trail** *Ys*" in block B3.

7. *Code Elimination*:

Since **no_trail**(*Ys*) is true immediately before the instruction "**trail** *Ys*" in block B3, this instruction can be deleted.

8. *Propagating* **no_trail**(...) *Assertions*:

Using rule NT4, it can be inferred that **no_trail**(*Ys*) is true immediately after the instruction "**cut_to** *CP*" in block B4. From this, two applications of rule NT3 allow us to infer that **no_trail**(*r2*) is true immediately after the instruction "*r2* := *Ys1*" towards the bottom of block B4. Then, an application of rule NT4 shows that **no_trail**(*r2*) is true immediately after the "**deallocate**" instruction in this block.

9. *Code Elimination*:

Recall that the Code Hoisting step at the beginning of the transformation introduced the instruction "**trail** *r2*" at the end of block B4. Since we have now inferred that **no_trail**(*r2*) is true at this point, this instruction can now be eliminated.

The resulting flow graph is shown in Figure 6(b). The transformed code involves one trail test, at the entry to *rem_dups*/2: the binding of new variables created in the body of the loop does not incur the overhead of extra trail tests, which is intuitively what is desired.

Note that it is not possible to proceed by simply introducing, at the beginning, a new entry node B in which *r2* is trailed, then propagating **no_trail**(*r2*) assertions downwards: the reason for this is that because of the ‘‘back edges’’ from blocks B3 and B4 (about which nothing is known at this point) into block B1, we cannot infer in this case that **no_trail**(*r2*) is true at the beginning of B0. It is for this reason that Code Introduction into B0, followed by hoisting, is necessary here.

4.4. Environment Allocation Reduction

This section describes two approaches to reducing the number of environments allocated at runtime. The first involves delaying the allocation of environments, while the second involves reusing an already allocated environment.

4.4.1. Environment Allocation Delaying

When the execution of a procedure begins in a Prolog program, it may not always be necessary to allocate an environment for that procedure on the stack. In the WAM, for example, parameter passing is done through registers, and if a clause can be executed using only register operations, i.e. if no space is used on the runtime stack, then the clause can be executed without allocating an environment.

When dealing with clauses that contain complex control-flow connectives, it may be the case that some execution paths in the clause require the allocation of an environment while others do not. In such cases, the simplest code generation strategy is to allocate an environment at the entrance to the clause. However, this is suboptimal if the execution path chosen does not require environment allocation. In this case, some redundant environment allocations may be eliminated using our transformation scheme. The hoisting transformation used here is *downward code hoisting*, and the only instruction considered for downward hoisting is the "**allocate**" instruction. The transformation is defined by the following:

Code Introduction: The code introduction transformation is not used here.

Code Elimination: This specifies that if the sequence of instructions "**allocate**; **deallocate**" occurs at any point within a basic block, it may be eliminated.

Code Replacement: This is used to move **allocate** instructions downward. The essential idea is that if a permanent variable *V* resides in a register *r* at the entry to the clause, then it may be possible to move an "**allocate**" instruction past an instruction containing references to *V*, provided that the reference to *V* is replaced by a reference to the register *r*. This requires a ‘‘symbol table’’ that gives the association between permanent variables and the registers that they began in. There are, of course, additional requirements that have to be satisfied before the transformation can be applied. The details of the transformation are as follows: **ST** is a set of ⟨*variable*, *register*⟩ pairs that is initialized to be empty. The transformation is driven by rules of the form given below (an exhaustive list is not given for reasons of space, but it is hoped that the reader will see the underlying idea and be able to complete the set of rules without much trouble):

(1) If a program point contains the instruction sequence *S*:

> **allocate**
> *Instrs*

where *Instrs* is an instruction sequence that does not alter or use the environment stack (e.g. does not refer to any permanent variable, and does not contain any instruction of the form "**call** ...", "**try** ..." etc.), then *S* can be transformed to

> *Instrs*
> **allocate**

(2) If a program point contains the sequence of instructions *S*:

> **allocate**
> **get_perm_var** *V*, *R*

and there is no variable V' such that $\langle V', R \rangle$ is in **ST**, then the instruction "**get_perm_var** *V*, *R*" can be deleted from *S* provided the pair $\langle V, R \rangle$ is added to **ST**.

(3) If a program point contains the sequence of instructions *S*:

allocate
get_perm_val *V*, *R*

and $\langle V, R' \rangle$ is in **ST** for some R', then *S* can be transformed to

get_temp_val R', *R*
allocate

and so on. Finally, when the "**allocate**" instruction cannot be migrated downward any further, unless the instruction immediately following the "**allocate**" instruction is "**deallocate**", an instruction

get_perm_var *V*, *R*

is introduced immediately after the **allocate** instruction for each pair $\langle V, R \rangle$ in **ST**.

The transformation strategy is to use downward code hoisting and replacement to move an "**allocate**" instruction down from the beginning of the clause. If it can be moved all the way down to a "**deallocate**" instruction, then the **allocate**/**deallocate** pair may be deleted. Even if this is not possible, however, delaying environment allocation can be advantageous, since in the transformed code, unification may fail before the "**allocate**" instruction is encountered, saving some work. This may be especially useful in highly nondeterministic ‘‘search’’-type applications, where execution tends to fail relatively often.

The savings realized from delaying environment allocation tend to be relatively small, because the bulk of the work in a program tends to be done along execution branches that require environments to be allocated anyway (though on some small programs, e.g. a non-tail-recursive factorial program and a program to test whether a term is ground, we observed speedups of over 10% from environment allocation reduction alone). The principal benefit of this transformation, in our experience, is that by delaying environment allocation, variables are kept in registers longer, enabling other optimizations, e.g. tag manipulation reduction, to be carried out more easily. An application of this transformation is given as part of an example considered in the next section.

4.4.2. Environment Reuse

Meier points out that one very often encounters tail recursive Prolog procedures that allocate an environment, process a term in some way, then deallocate the environment before making the tail recursive call [15]. This recursive call may then again allocate an environment and eventually deallocate it, and so on. This is illustrated by the following:

proc_list([], _, []).
proc_list([*X*|*Xs*], *Syms*, [*Y*|*Ys*]) :– *process*(*X*, *Syms*, *Y*), *proc_list*(*Xs*, *Syms*, *Ys*).

In a straightforward translation of this procedure, therefore, an environment would be allocated and deallocated at each iteration of the loop. In a conventional language, however, such a procedure would typically be written so that an environment would be allocated once at entry, updated at each iteration, and deallocated on exit. It is not possible to write the Prolog procedure directly in this way, because of the lack of iterative constructs such as `repeat` and `while`; however, we would like, wherever possible, to avoid penalizing the Prolog programmer for this, and reuse the environment for the procedure instead of repeatedly allocating and deallocating it.

The flow graph for the predicate *proc_list*/3 defined above is given in Figure 7(a). The intent of this optimization is to transform a loop that repeatedly allocates and deallocates environments to one that allocates an environment once, updates this environment as necessary during iteration, then deallocates it at the end of the iteration. It is necessary, therefore, that the entire loop should be defined in terms of one environment, so if the predicate is defined via multiple clauses, then these have to be merged to a single clause that uses disjunctions. The transformation is then defined by the following:

Code Introduction: The instruction pair "**allocate**; **deallocate**" may be introduced at any point in a flow graph. (Strictly speaking, this preserves equivalence only if we assume infinite memory, since otherwise it is possible to imagine situations where the newly introduced **allocate** can result in a stack overflow where previously there was none. However, as used in the optimization described, the **allocate** instruction is hoisted away, and hence does not pose a problem in practice.)

Code Elimination: If the instruction sequence "**deallocate**; **allocate**" occurs at any progam point, it may be deleted.

Code Replacement: If the instruction sequence *S*:

I
allocate

occurs at a program point, where *I* is a sequence of instructions that does not alter or use the environment stack, then *S* can be transformed to

allocate
I

Since the idea behind the optimization is to allocate an environment initially and update it as necessary during iteration, the first idea that suggests itself is to introduce a pair of instructions "**allocate**; **deallocate**" at the beginning of block B1 in Figure 7(a) – clearly, this is a correct transformation – and then

FIGURE 7

hoist the **allocate** instruction from blocks B1 and B2 into B0. The problem with this approach is that it results in the allocation of an environment at every call to *proc_list*/3, regardless of whether this is necessary or not. Given that cheap procedure calls are a very important feature of high-performance Prolog implementations (e.g. see [29]), this is undesirable and should be avoided. This can be done by splitting nodes B0 and B1: in effect, what this does is to create two distinct components in the flow graph, one of which does not allocate any environments, and the other which does; a call to the procedure selects one or the other component at entry, and thereafter stays in the selected component. We note, however, that the ultimate aim of the optimization is to move the **allocate** instruction out of the loop, and if this were attempted at this point, e.g. by introducing an "**allocate**; **deallocate**" pair at the top of block B1 and hoisting, then an environment would be allocated at every call to the predicate, which − as noted above − is undesirable. The simplest remedy to this problem is to split the edge from node B0 to B2. After this, the transformation proceeds as follows:

1. *Code Introduction*:

   The instruction sequence "**allocate**; **deallocate**" is introduced at the top of node B4, which was obtained by splitting the exit node B1.

2. *Code Hoisting*:

   The **allocate** instruction is hoisted from the tops of nodes B2 and B4.

3. *Code Hoisting*:

   The **allocate** instruction introduced into node B3 in the previous step is hoisted into node B2.

4. *Code Elimination*:

   The instruction pair "**deallocate**; **allocate**" at the bottom of node B2 is deleted.

The resulting flow graph is given in Figure 7(b). It accomplishes exactly what was desired: if the procedure has to allocate an environment, then it is allocated once at the entry to the loop, updated at each iteration, and deallocated before exit from the procedure. Note also that if there are loop-invariant computations in the body of the loop, these may be moved out of the loop at the end of this transformation (in the example above, it is tempting to move the assignment "*Syms* := *r2*" outside the loop, but this can be done only if additional information is available regarding the usage of registers within the predicate *process*/3). Moreover, compilers for traditional languages typically justify code motion out of loops on the grounds that most loops are executed at least once on the average: in loops that are often not executed (e.g. loops to skip whitespace in the lexical analysis component of compilers), invariant code motion out of a loop can actually be a ‘‘pessimization’’. However, this is not the case with the transformation described here, because the initial splitting transformations serve to insulate the ‘‘no iteration’’ case from any code that is moved out of the loop.

4.5. Bounds Check Reduction

The Prolog builtin *arg*/3 can be used to access any specified argument of a compound term. In most implementations, this can be done in $O(1)$ time, and hence is commonly used in programs that manipulate arrays, records and trees. For example, a predicate that checks whether a term is ground might be written as

ground(X) :–
    nonvar(X), (*atomic*(X) $\rightarrow$ *true* ; (*functor*(X, _, N), *ground_args*(N, X))).

ground_args(N, X) :–
    N =:= 0 $\rightarrow$ *true* ; (*arg*(N, X, T), *ground*(T), *N1* is $N-1$, *ground_args*(*N1*, X)).

However, a closer examination indicates that *arg*/3 performs many more operations than are involved in indexed access to a structure in a conventional language, and hence is significantly more expensive: executing the goal *arg*(N, T, X) involves the following operations:

(1) check the tag of T to ensure that it is bound to a constant or structure;

(2) check the tag of N to ensure that is bound to an integer;

(3) check that $N > 0$;

(4) look up the symbol table to retrieve the arity A of T;

(5) check that $N \leq A$;

(6) compute the address of the $N^{\text{th.}}$ argument of T;

(7) retrieve the $N^{\text{th.}}$ argument of T;

(8) unify the $N^{\text{th.}}$ argument of T with X.

Many of these computations become redundant when successive arguments of a term are accessed in a loop, as in the *ground_args*/2 example above. In this case, for example, operations (1) and (4) above are loop invariant computations, and can be moved out of the loop; and operation (2), and the tag manipulation implicit in operations (3) and (5), can be eliminated from the body of the loop using the transformation to reduce tag manipulation discussed earlier. However, this still leaves a significant amount of overhead in the task of accessing an argument of a term. This section considers how part of this overhead, namely part or all of the bounds checks, can be eliminated. In the *ground_args*/2 predicate above, for example, it is easy to see that in any call

?- . . ., *ground_args*(N, T), . . .

if N exceeds the arity of the term T, then this is detected right away, and the call fails; while if N does not exceed the arity of T, then this is verified in the first iteration, and since the value of the first argument of N decreases in subsequent iterations of the loop, further checking of the index against the upper bound is unnecessary.

This optimization can be handled as an instance of our transformation scheme. The expression "$t[i]$" denotes the $i^{\text{th.}}$ argument of the term referenced by t, retrieved without performing any bounds checking. Thus, a literal "*arg*(I, T, X)" is translated to the instruction sequence

if $I < 1$ **then** *fail*
ub := *arity*(T)

if $I > ub$ **then** *fail*
unify X, $T[I]$

where, for the sake of simplicity, the tag manipulation operations have been omitted. The transformation is defined by the following:

Code Introduction: If, at a point in a program, it can be guaranteed that the value of a variable x is a number N satisfying $N \geq LB$ for a known constant LB, then the instruction "**if** $x < LB$ **then** *fail*" can be inserted at that point without affecting the behavior of the program. Similarly, if it can be guaranteed that the value of x is a number N satisfying $N \leq UB$ for a known constant UB, then the instruction "**if** $x > UB$ **then** *fail*" can be inserted at that point without affecting the behavior of the program.

Code Elimination: In this case, code elimination cannot be made based on purely local considerations. It is given by the following: let π be a (prespecified) condition, and consider a point p in the basic block B under consideration such that

(*i*) $\pi(x)$ is never true at the entry to block B; and

(*ii*) B contains an instruction I : "**if** $\pi(y)$ **then** *fail*" at the point p, such that $\pi(y)$ is true at p only if $\pi(x)$ is true at the entrance to B.

Then, the instruction I can be deleted from B without affecting the behavior of the program.

Condition (*i*) may be established, for example, by checking that every path consisting only of success edges, from the entry node of the flow graph to the entrance to the block B, contains an instruction "**if** $\pi(x)$ **then** *fail*", such that x is not redefined between this instruction and the entry to B. While verifying the relationship required by condition (*ii*) between the conditions $\pi(y)$ and $\pi(x)$ may be difficult in general, simple special cases can be given that cover most commonly encountered situations. Two such special cases are: (*i*) $\pi(x)$ is of the form '$x > c$' or '$x >= c$', where c is a constant, and $x \geq y$; and (*ii*) $\pi(x)$ is of the form '$x < c$' or '$x =< c$', where c is a constant, and $x \leq y$. In either case, the relationship between x and y can usually be verified using classical flow analysis techniques to detect induction variables [1].

Code Replacement: This is given by the following: given a sequence of instructions S followed by an instruction

I : **if** $\pi(x)$ **then** *fail*

where S does not define x and does not contain any *call* instructions, I can be moved to the point immediately before S. (It should be noted that this transformation does not, strictly speaking, preserve equivalence, since it can improve the behavior of programs containing errors, in the sense that the transformed program can give rise to fewer runtime errors than the original program.)

The transformation proceeds in much the same way as before. This can be illustrated by considering the *ground_args*/2 example above. The flow graph before transformation is shown in Figure 8(a). First, the transformation for environment allocation reduction, discussed in the previous section, is performed: the **allocate** instruction is hoisted downward, followed by code replacement to migrate the **allocate** instruction down the basic blocks, and finally by the elimination of an **allocate**/**deallocate** pair in block B1. The transformations for bounds check reduction are then carried out as follows:

FIGURE 8

1. *Code Introduction*:

   It is straightforward to establish, by flow analysis, that the second argument to *ground_args*/2 is always a nonvariable term. Since the arity of a nonvariable term is necessarily nonnegative, it follows that the value of *r1* cannot exceed that of *ub*, so the instruction "**if** *r1* > *ub* **then** *fail*" can be introduced into B1.

   The result of code introduction, therefore, is to introduce, at the beginning of block B1, the instruction sequence "*ub* := *arity*(*r2*); **if** *r1* > *ub* **then** *fail*".

2. *Code Hoisting*:

   The instruction sequence "*ub* := *arity*(*r2*); **if** *r1* > *ub* **then** *fail*" is hoisted from B1 and B2 into B0.

3. *Code Hoisting*:

   The instruction sequence "*ub* := *arity*(*r2*); **if** *r1* > *ub* **then** *fail*" is hoisted again, this time from B0. Since B0 is the entry node for the procedure, a new entry node is created containing only these instructions. This step also results in these two instructions being introduced at the bottom of block B2.

4. *Code Elimination*:

   We observe that (*i*) the value of *ub* at the entry to block B2 is always the arity of the second argument to *ground_args*/2; (*ii*) every path consisting only of success edges from the entry node to the entry of block B2, there is an instruction "**if** *r1* > *ub* **then fail**", such that *r1* is not redefined between this instruction and the entry to B2; and (*iii*) the value of *r1* only decreases between the entry to B2 and the instruction "**if** *r1* > *ub* **then fail**" at the end of B2, which means that if *r1* is greater than *ub* at the end of B2 then it must have been greater than *ub* at the entry to B2. Thus, the conditions for code elimination are satisfied, and the bounds check at the end of B2 can be eliminated. The instruction "*ub* := *arity*(*r2*)" at the end of block B2 is now dead code, and can also be deleted.

In the resulting flow graph, illustrated in Figure 8(b), the bounds check against the upper bound is performed only once, at the entry to the loop.

5. Pragmatic Considerations

The paper so far has discussed a low level code transformation scheme that can be instantiated in different ways to obtain different kinds of specific low level optimizations. For the specific optimizations so obtained, it is usually the case that the code introduction and hoisting transformations follow specific patterns that are easy to identify. Such patterns may be taken advantage of to realize more efficient implementations of these optimization algorithms. However, it is difficult to specify a general algorithm for applying such transformations, because a great deal depends on specific features of Code Introduction and Code Elimination, which vary from one optimization to the next.

The transformations discussed perform code introduction and hoisting in the hopes of eventually realizing a code elimination step. One simple way to guide the transformation, therefore, is to ensure that code elimination will be possible before applying the transformation. This can be done using *reaching definitions*: a definition *d* of a variable *x* is said to *reach* an instruction *s* if there is an execution path from *d* to *s* along which *x* is not redefined. Sets of reaching definitions can be obtained using classical dataflow analysis techniques [1]. Once these have been computed, a program point that is being considered for the introduction or hoisting of a sequence of instructions is tested to see whether the set of definitions that reach that point suggest that code elimination will eventually be possible. The transformations are carried out only if this is found to be the case. For example, in tag manipulation reduction, the flow graph is first tested to see if a "**tag** *u t*" instruction can reach an "**untag** *u t*" instruction along the back edge of a loop: if there is no such reaching definition, the transformation is not considered further at that point.

Another important consideration is the code introduction step, which opens up avenues for code hoisting and the eventual code elimination. This must be performed with some care in order to avoid slowing down the program by introducing code inside loops. Node splitting and edge splitting can be used to allow hoisting to be performed without introducing code inside loops. Since the sole purpose of code introduction is to allow code hoisting to be carried out, it is necessary to define which basic blocks need to be taken into account when considering code hoisting from a block *B*. This is given by the *siblings* of *B*, defined as follows:

Definition: Given a basic block *B*, a basic block *Q* is a *sibling* of *B* if (*i*) there is a basic block *P* that is a predecessor of both *B* and *Q*; or (*ii*) there is a basic block *R* such that *Q* is a sibling of *R* and *R* is a sibling of *B*. ♦

It is not difficult to see that if the block *B* starts with a sequence of instructions *S*, then *S* can be hoisted from *B* into the predecessors of *B* if and only if every sibling of *B* starts with *S*. Let $sibs_Y(B)$ denote those siblings of *B* that also start with the instruction sequence *S*, and $sibs_N(B)$ denote those siblings of *B* that do not start with this sequence. In order to perform hoisting, it is necessary to introduce *S* at the beginning of each block in $sibs_N(B)$, by means of code introduction (provided that the preconditions for this are satisfied). Suppose that the code introduction transformation adds an instruction sequence S' at the beginning of each block in $sibs_N$. If the cost of the instruction sequence *S* is *C*, and that of $S' - S$ (the suffix of S' left over after hoisting *S*) is C', then code introduction should be performed only if there is a net savings realized, i.e. if

$$C * \sum_{P \in sibs_Y(B)} freq(P) \quad > \quad C' * \sum_{P \in sibs_N(B)} freq(P)$$

where $freq(B)$ is the expected frequency of execution of a basic block B. In general, of course, the estimation of execution frequencies is difficult. However, code of good quality can usually be generated by assuming ‘‘reasonable’’ values for the number of times the body of a loop is executed, e.g. assuming that each loop is executed five or ten times on the average (experience with compilers for traditional languages suggests that this works quite well in practice, e.g. see [20]).

Based on such a strategy for estimating the execution frequency of a loop, one may use the following general approach towards applying these transformations:

(1) Identify the target instructions I to be eliminated, with priority given to instructions within innermost loops.

(2) If code elimination requires that the instructions I be juxtaposed with some other instructions I', then verify that I and I' are (potentially) juxtaposable using reaching definitions. Here, consider also the possibility of creating an instance of I' in a sibling block via code introduction.

(3) If there are several possible blocks B_i that are sources of the instructions I' necessary for code elimination, choose one that is as ‘‘close’’ to the basic block B containing I as possible, in the following sense: let this chosen block be B', and let the basic block that is the nearest common ancestor of B and B' be A, then try to minimize the distance between A and B, and between A and B'. Let this distance be n, then n hoisting steps may be necessary before code elimination can be carried out.

(4) Apply code hoisting and code replacement upto n times, checking at each hoisting step to ensure that the estimated cost of the flow graph, based on estimates for the execution frequencies of loops, is not increased (modulo any code elimination that may become applicable). If it appears that a hoisting step will increase the cost of the program, this must be because code is being hoisted from a block with a lower execution frequency into one with a higher execution frequency, i.e. from the outside to the inside of a loop. In such cases, it is possible to either prevent the hoisting of code into the loop by applying node splitting or edge splitting, as illustrated in the Environment Reuse optimization of Section 4.4.2, or to abort the transformation entirely.

(5) Carry out code elimination.

In general, it may be possible to determine whether a particular sequence of hoisting steps can be carried out without introducing code into loops by ‘‘calculating ahead’’, without actually carrying out the transformations. This can improve the efficiency of the transformations. Moreover, there may be more than one instruction sequence I targeted for elimination, and in general the transformations for these may be carried out together. However, it should be noted that in some cases, carrying out the transformations for different optimizations at the same time may result in a failure to eliminate some instructions, even though they would have been eliminated if the optimizations had been carried out one after the other in the appropriate order. The reason for this is that, as illustrated by the examples of Section 4, the Code Replacement transformations are often subject to conditions that specify that a register or variable not be used or defined within some instruction sequence. It may happen, depending on the particular optimizations being considered, that if they were carried out in sequence, then code elimination from preceding optimizations would delete certain uses or definitions of a variable or register, allowing a subsequent optimization to proceed, but that this does not happen if the optimizations are carried out ‘‘concurrently’’. Note that this also implies that the applicability of an optimization on a given program may depend on what other optimizations have already been applied.

6. Experimental Results

Experiments were run on SB-Prolog on a Vax-8650 to gauge the efficacy of the transformations discussed for reducing redundant tag manipulation, dereferencing, trail testing, and bounds checking. We caution the reader that the numbers reported pertain, necessarily, to one implementation on one machine (in particular, the overhead of byte code interpretation in SB-Prolog may distort some of the speedup figures): speedups from such optimizations may be different on other implementations on other machines. Nevertheless, it is our opinion that these numbers may serve at least as a ‘‘plausibility test’’ for the optimizations discussed earlier.

When testing the improvements resulting from tag manipulation reduction, we deliberately chose a set of programs that performed a great deal of integer tag manipulation: our objective was to see what sort of performance improvements might be obtained under favorable circumstances. The only objects considered for tag stripping in our experiments were integers, and the additional instructions introduced to deal with untagged operands were those for arithmetic and relational operators. The programs tested were the following: *factorial*, a tail recursive factorial program; *tr_fib*, a tail recursive program to compute fibonacci numbers; *fibonacci*, a linear recursive (but not tail recursive) program to compute fibonacci numbers; *nth_element*, a program to extract a specified element of a list (in our experiment, the last element of a list of 50 elements); and fourqueens. As the figures in Table 1 indicate, the performance improvements range from 8% to 48%, which is quite encouraging. This suggests that even better performance gains are possible, by also considering objects other than integers and compiling to native code. It should be noted, on the other hand, that machines with hardware tag support may incur far less overhead for operations on tagged objects, with correspondingly smaller improvements resulting from this optimization [25].

The programs used to test improvements resulting from the reduction of redundant dereferencing were *factorial*, *tr_fib*, *nth_element* and *fibonacci*. The speed improvements in this case, which are given in Table 2, ranged from 3% to 6%.

The test improvements from trail test reduction, the programs used were the *factorial*, *tr_fib*, and *fibonacci* programs from dereference removal testing, together with *rem_dups*, a program that removes all duplicates from a list of length 50, and *nrev*, the naive reverse program. The speed improvements, given in Table 3, range from 0 in the case of *fibonacci* (where there is exactly one trail test, at the very end of the loop, so that there is no net reduction in the number of trail tests due to this optimization) to 5.5% for *tr_fib*.

The programs used to test improvements resulting from the reduction of redundant bounds checks were the following: *ground*, a program to test whether a term is ground; *subst*, a program that, given terms t_1, t_2 and t_3, returns the term obtained by substituting each occurrence of t_1 within t_2 by t_3; *subsumes*, a program to check whether one term subsumes another; *array_upd*, a program to update an element of a given array (tested with an array of size 256, organized as a balanced quadtree of depth 4); and *mat_mult*, a program to multiply two matrices (tested with two 50×50 matrices). The results of our experiments are given in Table 4. The improvements in this case are disappointingly small (Markstein et al. report that static elimination of bounds checks in imperative languages like PL/I can result in a 7-10% decrease in the number of instructions executed [14]). This is due at least in part to the overhead of byte code interpretation in SB-Prolog, and suboptimal use of hardware registers (compared to similar programs in Fortran or PL/I), which tend to swamp the improvements due to the elimination of redundant bounds checks; we expect better speedups from this optimization in systems that have smaller byte code interpretation overhead, or that compile to native code. Apart from the byte code interpretation overhead, the programs tested tend to do significant amounts of other operations, such as unification and procedure calls, which are absent in comparable programs in PL/I or Fortran: the overhead incurred in these operations also dilute the speedups measured from this optimization.

Finally, we tested the combined effects of three optimizations – reduction of redundant tag manipulation, environment allocation and bounds checks. The results are given in Table 5.

7. Conclusions

Most of the research to date on improving the efficiency of Prolog programs has focussed on high-level transformations. However, these have the shortcoming that they cannot express implementation level details, and hence cannot address low level optimizations. This paper describes a simple code improvement scheme that can be used to specify and reason about a variety of low level optimizations. Because the optimization-specific transformations typically involve only local reasoning, they are relatively easy to implement and verify. Applications illustrated include the reduction of reduntant tag manipulation operations, dereferencing, trail testing, environment allocation, and bounds checking.

Acknowledgements

Mats Carlsson made many helpful comments on an earlier version of this paper. Comments by the anonymous referees helped improve the contents and presentation of the paper significantly.

References

1. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers - Principles, Techniques and Tools*, Addison-Wesley, 1986.

2. A. Appel, Runtime Tags Aren't Necessary, Tech. Rep. CS-TR-142-88, Dept. of Computer Science, Princeton University, Princeton, NJ, Mar. 1988.

3. M. Carlsson, On Compiling Indexing and Cut for the WAM, Research Report R86011, Swedish Institute of Computer Science, Spanga, Sweden, Dec. 1986.

4. M. Carlsson, personal communication, Nov. 1989.

5. S. K. Debray and J. C. Peterson, Compile-time Tag Stripping, Unpublished manuscript, Dept. of Computer Science, The University of Arizona, Tucson, Nov. 1988.

6. S. K. Debray, Static Inference of Modes and Data Dependencies in Logic Programs, *ACM Transactions on Programming Languages and Systems 11*, 3 (1989), pp. 419-450.

7. S. K. Debray, Flow Analysis of Dynamic Logic Programs, *J. Logic Programming 7*, 2 (Sept. 1989), pp. 149-176.

8. D. Jacobs and A. Langen, Accurate and Efficient Approximation of Variable Aliasing in Logic Programs, in *Proc. North American Conference on Logic Programming*, Oct. 1989, pp. 154-165. MIT Press.

9. G. Janssens and M. Bruynooghe, An Instance of Abstract Interpretation Integrating Type and Mode Inferencing, in *Proc. Fifth International Conference on Logic Programming*, Seattle, Aug. 1988, pp. 669-683. MIT Press.

10. G. Janssens, Deriving Run Time Properties of Logic Programs by Means of Abstract Interpretation, PhD Dissertation, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1990.

11. T. Lindholm, personal communication, Nov. 1989.

12. M. J. Maher, Correctness of a Logic Program Transformation System, Research Report RC 13496, IBM T. J. Watson Research Center, Yorktown Heights, NY, May 1988.

13. A. Marien, G. Janssens, A. Mulkers and M. Bruynooghe, The Impact of Abstract Interpretation: an Experiment in Code Generation, in *Proc. Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989. MIT Press.

14. V. Markstein, J. Cocke and P. Markstein, Optimization of Range Checking, in *Proc. ACM SIGPLAN '82 Symposium on Compiler Construction*, Boston, June 1982, pp. 114-119. SIGPLAN Notices vol. 17 no. 6..

15. M. Meier, Recursion vs. Iteration in Prolog, in *Proc. Second NACLP Workshop on Logic Programming Architectures and Implementations*, Austin, TX,, Nov. 1990, pp. 26-35.

16. M. Meier,, Compilation of Compound Terms in Prolog, in *Proc. 1990 North American Conference on Logic Programming*, MIT Press., Austin, TX, Nov. 1990, pp. 63-79.

17. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.

18. K. Muthukumar and M. Hermenegildo, Determination of Variable Dependence Information through Abstract Interpretation, in *Proc. North American Conference on Logic Programming*, Oct. 1989, pp. 166-185. MIT Press.

19. A. Pettorossi and M. Proietti, Decidability Results and Characterization of Strategies for the Development of Logic Programs, in *Proc. Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989. MIT Press.

20. M. L. Powell, A Portable Optimizing Compiler for Modula-2, in *Proc. SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp. 310-318.

21. H. Seki, Unfold/Fold Transformation of Stratified Programs, in *Proc. Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989. MIT Press.

22. H. Tamaki and T. Sato, Unfold/Fold Transformations of Logic Programs, in *Proc. 2nd. Logic Programming Conference*, Uppsala, Sweden, 1984.

23. J. Tan, Prolog Optimization by Removal of Redundant Trailings, Technical Report, Dept. of Computer Science, National Taiwan University, Taipei, April 1989.

24. A. Taylor, Removal of Dereferencing and Trailing in Prolog Compilation, in *Proc. Sixth International Conference on Logic Programming*, June 1989, pp. 48-60. MIT Press.

25. A. Taylor, LIPS on a MIPS: Results from a Prolog Compiler for a RISC, in *Proc. Seventh International Conference on Logic Programming*, MIT Press, Jerusalem, June 1990, pp. 174-188.

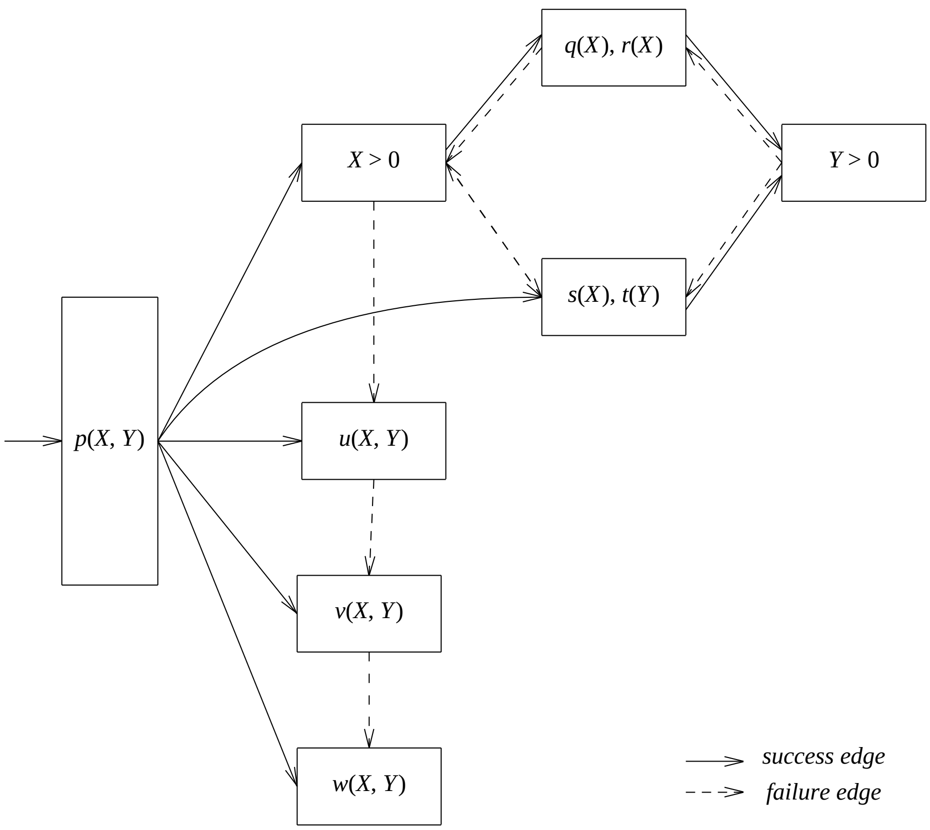
26. A. K. Turk, Compiler Optimizations for the WAM, in *Proc. 3rd. International Conference on Logic Programming*, London, July 1986, 410-424. Springer-Verlag LNCS vol. 225.

27. P. Van Roy, B. Demoen and Y. D. Willems, Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and Determinism, in *Proc. TAPSOFT 1987*, Pisa, Italy, Mar. 1987.

28. P. Van Roy and A. M. Despain, The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, in *Proc. 1990 North American Conference on Logic Programming*, MIT Press, Austin, TX, Oct. 1990, pp. 501-515.

29. P. Van Roy and A. M. Despain, The Making of the Aquarius Prolog System, in *Proc. NACLP-90 Workshop on Logic Programming Architectures and Implementations*, Austin, TX, Nov. 1990, pp. 11-15.

30. D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.



$p(X, Y)$:– ($X > 0 \rightarrow (q(X), r(X))$; $(s(X), t(Y))), Y > 0$.
$p(X, Y)$:– $u(X, Y)$; $v(X, Y)$.
$p(X, Y)$:– $w(X, Y)$.

Figure 1: Example of an augmented flow graph for a predicate

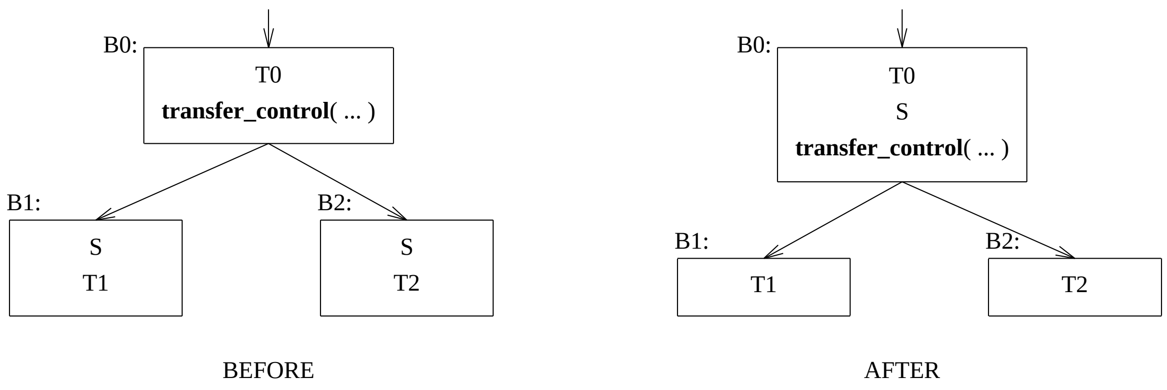


Figure 2 : The Upward Code Hoisting Transformation

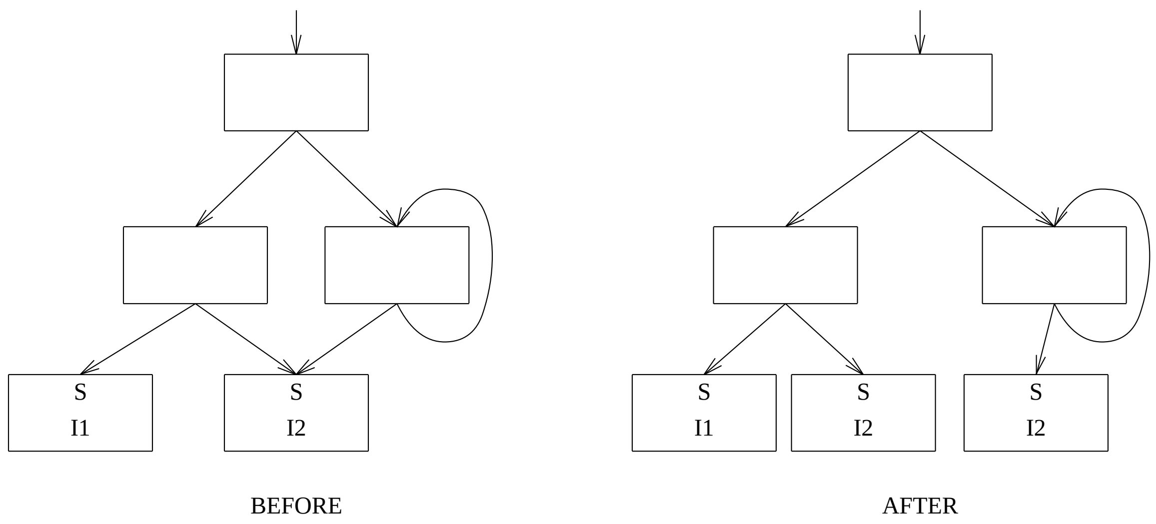


Figure 3 : Node Splitting

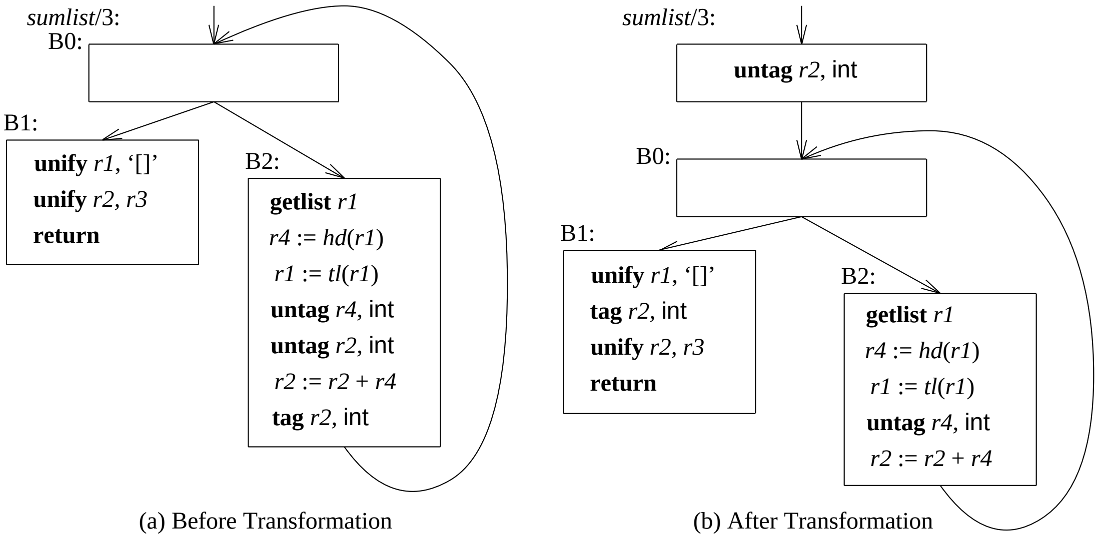


Figure 4: Reducing Redundant Tag Manipulation Operations

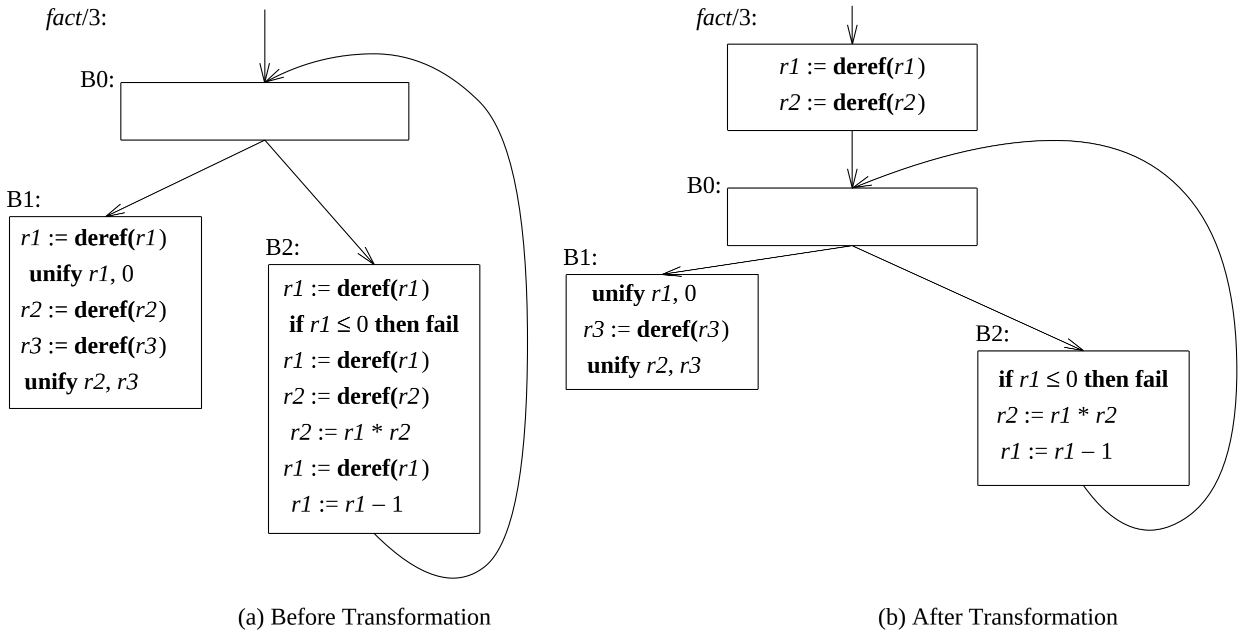
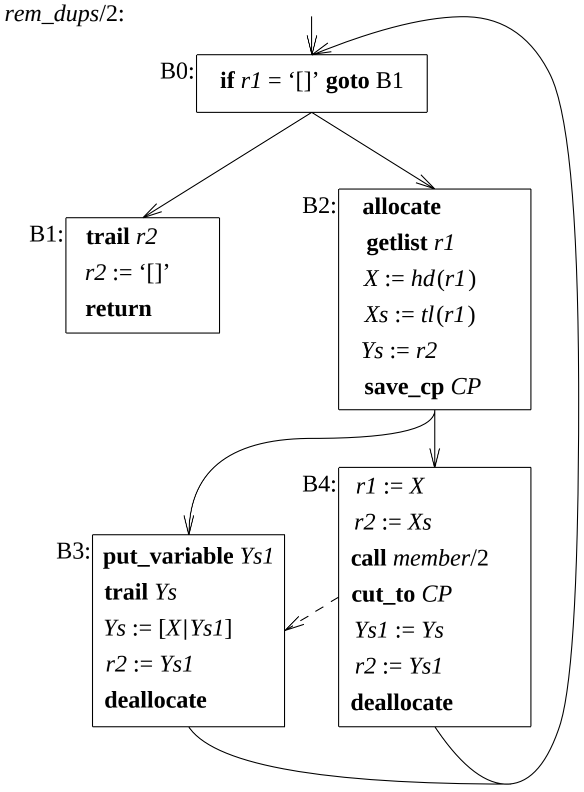
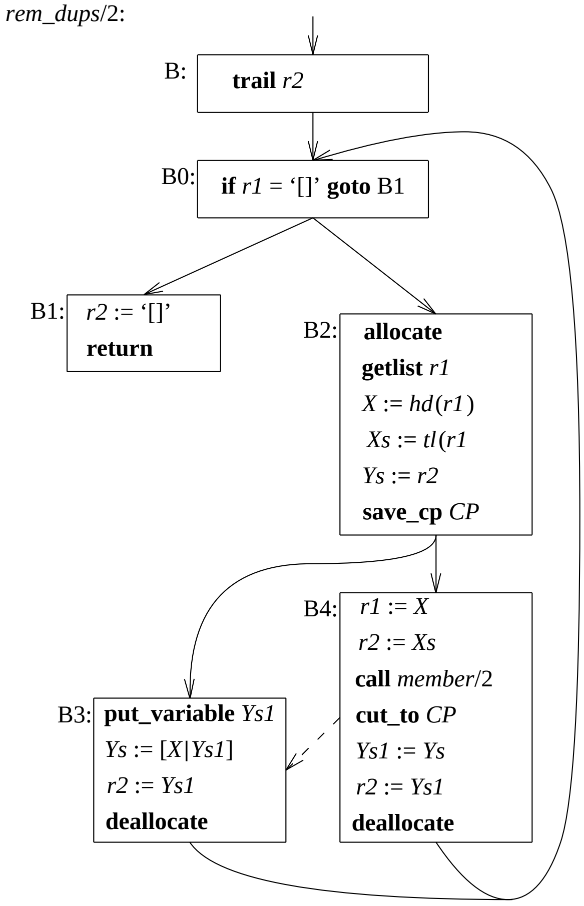


Figure 5: Reducing redundant Dereference operations.



(a) Before Transformation

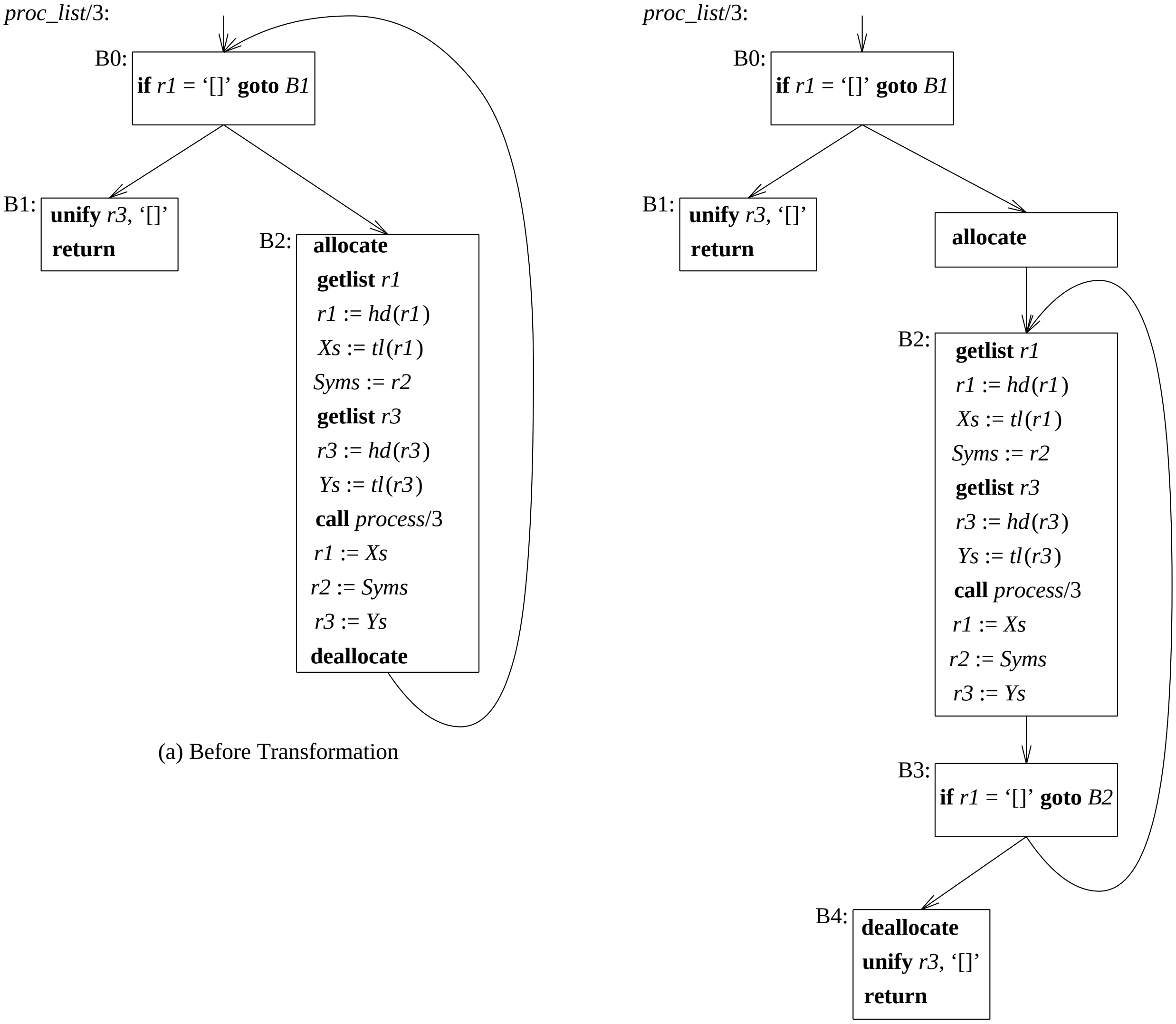


(b) After Transformation

success edge

failure edge

Figure 6 : Reducing Redundant Trail Tests



(a) Before Transformation

(b) After Transformation

Figure 7 : Environment Reuse

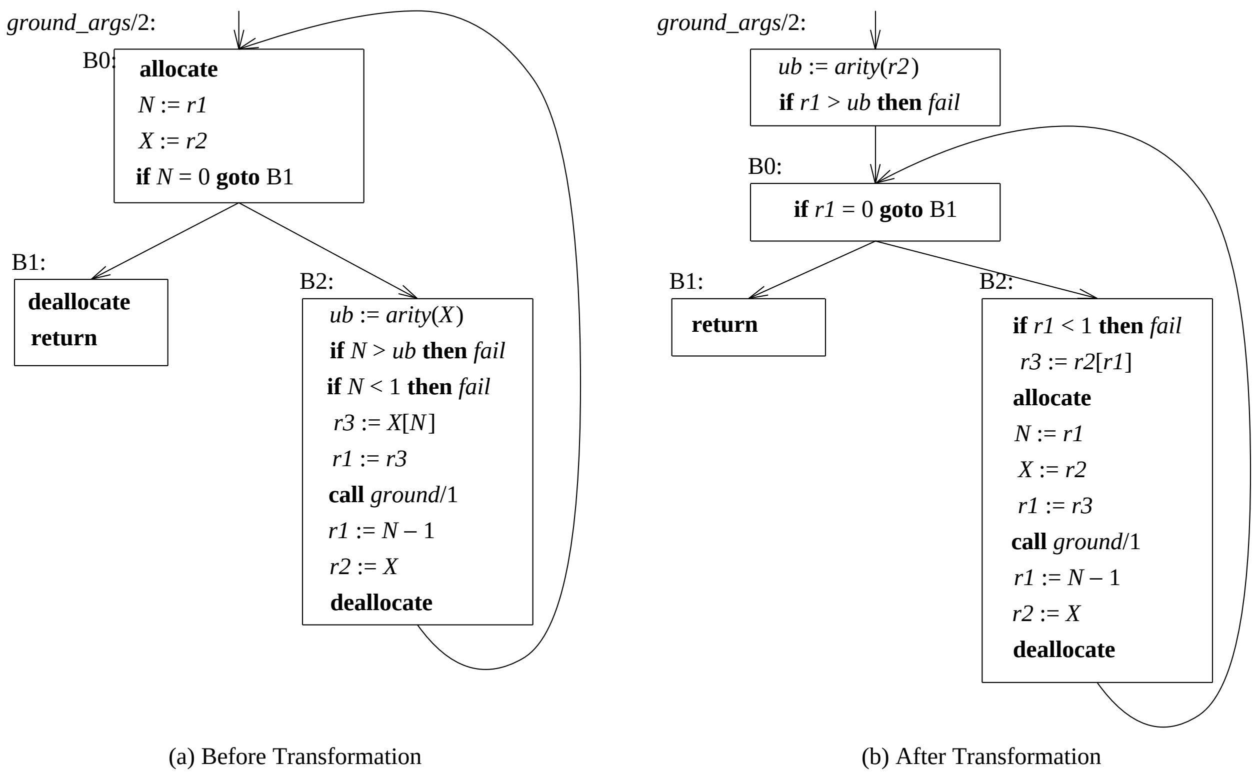


Figure 8: Reducing redundant Bounds Check operations.

Program	Iterations	pre-optimization	post-optimization	% Δ
factorial(10)	100000	27.12 secs	14.04 secs	48.2
tr_fib(40)	30000	34.97 secs	21.14 secs	39.5
nth_element	10000	21.24 secs	15.60 secs	26.6
fibonacci(40)	10000	24.72 secs	20.54 secs	16.9
fourqueens	1000	13.07 secs	12.02 secs	8.0

Table 1 : Experimental speedups due to tag manipulation reduction

Program	Iterations	pre-optimization	post-optimization	% Δ
tr_fib(30)	25000	25.81 secs	24.24 secs	6.1
factorial(10)	100000	33.60 secs	31.88 secs	5.1
nth_element	25000	39.99 secs	38.46 secs	3.8
fibonacci(30)	25000	62.23 secs	60.29 secs	3.1

Table 2 : Experimental speedups due to dereferencing reduction

Program	Iterations	pre-optimization	post-optimization	% Δ
tr_fib(30)	25000	25.14 secs	23.76 secs	5.5
fibonacci(30)	25000	54.32 secs	52.03 secs	4.2
nrev	5000	44.34 secs	43.55 secs	1.8
rem_dups	5000	96.08 secs	95.68 secs	0.4
factorial(10)	100000	32.93 secs	32.93 secs	0

Table 3 : Experimental speedups due to trail test reduction

Program	Iterations	pre-optimization	post-optimization	% Δ
mat_mult	1	34.40 secs	33.72 secs	1.98
ground	100	42.97 secs	42.53 secs	1.0
array_upd	5000	32.27 secs	32.00 secs	0.8
subst	500	57.72 secs	57.43 secs	0.5
subsumes	50	47.33 secs	47.11 secs	0.4

Table 4: Experimental speedups due to bounds check reduction

Program	Iterations	pre-optimization	post-optimization	% Δ
mat_mult	1	34.40 secs	23.56 secs	31.5
ground	100	42.97 secs	31.02 secs	27.8
subst	500	57.72 secs	51.09 secs	11.5

Table 5: Speedups due to combined optimizations