Compiler Optimizations for Low-level Redundancy Elimination: An Application of Meta-level Prolog Primitives

Saumya K. Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA

Abstract

Much of the work on applications of meta-level primitives in logic programs focusses on high-level aspects such as source-level program transformation, interpretation, and partial evaluation. In this paper, we show how meta-level primitives can be used in a very simple way for low-level code optimization in compilers. The resulting code optimizer is small, simple, efficient, and easy to modify and retarget. An optimizer based on these ideas is currently being used in a compiler that we have developed for Janus [6].

1 Introduction

Much of the work on applications of meta-level primitives in logic programs focuses on high-level aspects such as source-level program transformation, interpretation, and partial evaluation. In this paper, we consider instead the use of meta-level Prolog primitives in low-level code optimization in compilers. We show how such primitives can be used in a very simple way for a low-level code optimization called *common subexpression elimination*. The resulting code optimizer is small, simple, efficient, and easy to modify and retarget. Because it requires only a very simple logical description of the instruction set under consideration, correctness is simple to guarantee.

The application we describe is not profound—indeed, its primary appeal to us is its simplicity (it happens also to be efficient, effective, and easy to implement). An experienced Prolog programmer might very well consider it to be an "obvious hack," and if the only application we could find for it was in Prolog compilers written in Prolog, then we would consider it to be too narrow an application to be much more than a curiosity. One of the contributions of this paper is to show that these ideas are applicable, not only to the compilation of logic programming languages, but also to compilers for traditional functional and imperative languages. A related technique is dicussed by Komorowski in the context of partial evaluation [8].

Conceptually, the optimization algorithm described here can be thought of as a scheme to transform an intermediate representation of a program from a tree to a DAG via some sort of *value numbering* scheme [1]. This, however, will be true for essentially every algorithm for common subexpression elimination. First, almost any scheme for common subexpression elimination can be thought of in terms of merging distinct computations ("nodes") that are equivalent in some sense, i.e., in terms of a transformation from a tree to a DAG. Moreover, unless the implementation is entirely naive, checking whether two instruction sequences represent the computation of a common subexpression will be carried out, wherever possible, without exhaustively comparing the two sequences: this can typically be formulated in terms of some kind of value numbering scheme. What we feel is interesting and elegant about the approach described in this paper is that the use of unification, together with meta-level primitives, allows much of the low-level clutter associated with these operations to be avoided by the compiler writer. As a result, the code generator produced is transparent, easy to understand, verify, modify, and retarget.

2 Background

2.1 Common Subexpressions

During compilation, a program is typically translated from the source language to a lower level intermediate language. The program resulting from the translation to intermediate code may contain *common subexpressions*. An occurrence of an expression E is called a common subexpression if E was previously evaluated, and the values of variables occurring in E have not changed since the previous computation [1]. Common subexpressions may arise in a program either because there are multiple occurrences of an expression in the source program (which may happen, for example, after macro expansion), or because the translation to intermediate code makes explicit lower-level operations that are not visible at the source level. As an example, the Fortran code fragment

```
A[i] = A[i] + 1
```

may translate, on a machine with 4-byte words, to the intermediate code sequence

```
t0 := 4*i
t1 := A[t0]
t2 := t1+1
t3 := 4*i
A[t3] := t2.
```

Here, the expression `4*i` is a common subexpression: it arises because the low-level details of array subscripting are not visible at the source level.

In general, the performance of a program can be improved by eliminating code to recompute common subexpressions, and using the previously computed value instead. In practice, common subexpression elimination incurs a cost, since it may tie up a machine register to hold the value of the expression for subsequent uses, or result in stores and loads of the saved value from memory. In general, a compiler has to weigh the savings realized from common subexpression elimination against the costs incurred to determine whether a particular common subexpression is worth eliminating (e.g., see [14]). Thus, common subexpression elimination involves

1. identifying common subexpressions;
2. deciding which of these are worth eliminating; and
3. transforming the code sequence to eliminate such common subexpressions.

Since these three points are essentially orthogonal to each other, we will focus on points (1) and (3) in this paper.

2.2 Static Single Assignment Form

As we will see, our compilation model views a variable as a logical entity that can be defined at most once. In an imperative language, it may happen that the source program contains multiple assignments to a variable. It is possible to transform such programs to *static single assignment form* [4, 5] to conform to our compilation model. In static single assignment form, a program is transformed so that the program text contains at most one assignment to any variable along any execution path. The significance of this transformation stems from the fact that (even in imperative languages) single-assignment source variables are desirable for a number of optimizations, including parallelism detection [4, 10, 13]. Dynamically, a program with loops may assign many times to the same variable, even if only one assignment appears in the program text (a scheme to get around this problem, involving the creation of new variables dynamically, is discussed in [4]). In this section, we briefly review the transformation of a program to static single assignment form, as described in [5].

First, consider the transformation for a single basic block with multiple definitions of a variable. The transformation simply renames all definitions of the variable except for the final one, and their corresponding uses, as illustrated below:

Original		**Renamed**	
`X = ...`	`/* define X */`	`X1 = ...`	`/* define X1 */`
`...`		`...`	
`= X`	`/* use X */`	`= X1`	`/* use X1 */`
`...`		`...`	
`X = ...`	`/* define X */`	`X2 = ...`	`/* define X2 */`
`...`		`...`	
`= X`	`/* use X */`	`= X2`	`/* use X2 */`
`...`		`...`	
`X = ...`	`/* define X */`	`X = ...`	`/* define X */`

Since only the last definition of a variable within a block can be used outside that block, this transformation does not affect any use of that variable in other basic blocks. Within a basic block, each use of a variable must be renamed, if necessary, to match the corresponding definition.

Next, consider a case where multiple definitions reach a use. In this case, we first identify the "join birthpoints" of variables in the control flow graph, i.e., points in the control flow graph where several definitions of a variable meet on different incoming edges for the first time. Consider a join birthpoint for a variable `X` where k paths, each with a definition for `X`, meet. We transform the program by renaming `X` along each path, ensuring that this renaming does not introduce the same name along different paths. Suppose that for each path i, $1 \leq i \leq k$, the last definition of `X` has been renamed in this process to $\mathtt{X}_i$. We then ensure that the original variable contains the correct value by adding an assignment $\mathtt{X} = \phi(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ for each path i, $1 \leq i \leq k$: here, $\phi(\ldots)$ is a special form of assignment, called a *join-definition*, that assigns the appropriate value depending on which branch of the conditional was taken: in practice, this can be implemented in a fairly straightforward way (see [4]). The approach is illustrated by the following example:

Original	**Renamed**

```
X = A+B                       X1 = A+B
if (...)                      if (...)
     then X = X+1                  then X2 = X1+1
          Z = 0                          Z = 0
     else X = X+2                  else X3 = X1+2
                              X = φ(X2, X3)
     ...                       ...
Y = 2*X                       Y = 2*X
```

This step eliminates multiple assignments to a variable in the absence of loops. A scheme to deal with loops, via dynamic creation of new variables, is described in [4]: since we will be concerned primarily with loop-free programs, this will not be discussed further here.

3 Common Subexpression Elimination: The Traditional Approach

Common subexpression elimination is usually carried out by analyzing the abstract syntax tree for a compilation unit (typically a procedure) to find identical expression subtrees. Suppose that E_1 and E_2 are two such identical subtrees, and E_1 is guaranteed to be evaluated before E_2. Then, if the variables in E_2 can be guaranteed to be unchanged since the evaluation of E_1, then the subtree E_2 can be replaced by a pointer to the subtree E_1. As a result, the syntax tree is transformed into a DAG in which nodes with more than one parent correspond to common subexpressions.

Now suppose we are processing an expression tree E in a procedure. In a naive implementation, determining whether there is another subtree elswehere in the procedure identical to E might be carried out by actually matching E against the various expression subtrees occurring in the procedure. This, of course, would be hopelessly inefficient in general, even if the search is restricted to "previously computed" expression subtrees. In practice, therefore, a more sophisticated scheme called *value numbering* [3, 9, 10] is used. The essential idea is to assign special symbolic names called value numbers to expressions. Then, if two expressions $E_1 \equiv op_1(t_1, \ldots, t_n)$ and $E_2 \equiv op_2(u_1, \ldots, u_n)$ satisfy (i) $op_1 = op_2$, and (ii) the value number of t_i is the same as that of u_i, $1 \leq i \leq n$, then E_1 and E_2 are guaranteed to compute the same value.

The implementation of value numbering, however, can be considerably more complicated that this description might suggest. For example, [1] describes an implementation scheme that involves using a hash table to keep track of expressions that are potentially common subexpressions. Moreover, if there can be multiple assignments to a variable in the program, then this structure has to be kept consistent with updates.

4 Our Approach

4.1 The Compilation Model

The most significant difference between our compilation model, and that used in traditional compilers, is that we view a (source or temporary) variable as a logical entity whose value can be defined at most once. This turns out to simplify significantly the subsequent reasoning about, and optimization of, the intermediate code program, since there is no need to worry about the value of a variable changing due to multiple assignments to it.

Traditional compilers typically attempt to conserve machine resources by deallocating temporary variables when they are no longer needed, and reusing such deallocated temporaries later if possible. In our model, in contrast, no attempt is made to reuse temporary variables during intermediate code generation. The effects of such reuse are obtained later, during final code generation, when temporary variables are mapped to machine resources such as memory locations or registers: at this time, liveness information can be used to map variables with disjoint lifetimes into the same register or memory location. Multiple assignments to variables in the source program can be handled by transformation to static single assignment form, as discussed earlier.

We also assume that intermediate code instructions are *recyclable*, i.e., can be reused. The idea here is the following: suppose we have two instructions I_1 and I_2, with identical operands, in a basic block. From the assumption that variables and temporaries are single assignment entities, this means that these instructions will have identical operand values at runtime. The assumption of recyclability states that in this case, the result from the first instruction I_1 can be recycled and used in place of the second instruction I_2. While the assumption seems not too unreasonable, it need not be satisfied in practice, e.g., if the instruction under consideration is nondeterministic, or if it involves side effects, e.g., for I/O. In practice, however, our scheme can be used even if not all instructions in the language under consideration are recyclable, as long as we restrict our attention to recyclable instructions.

In the remainder of this paper, we consider common subexpression elimination in loop-free code fragments only (this is not as bad as it may seem, since most compilers restrict themselves to common subexpression elimination within basic blocks or extended basic blocks): in this case, the transformation to static single assignment form suffices to ensure that our assumptions are satisfied.

For the remainder of the paper, intermediate code instructions will be represented as follows unless explicitly mentioned: an instruction with opcode `op`, operands $In_1, \ldots, In_m$, and results $Out_1, \ldots, Out_n$ will be represented as

`op([`$In_1, \ldots, In_m$`], [`$Out_1, \ldots, Out_n$`])`.

For example, the instruction `add([R1, R2], [R3])` indicates that the sum of `R1` and `R2` is assigned to `R3`. If an instruction has no operands, then the first argument is the empty list `[]`: for example, an increment instruction might be written '`inc([], [X])`'. It is important to note that any entity that may be "read" by an instruction is expected to be listed explicitly as an operand, while any entity that may be "written" by an instruction is expected to be listed as a result: this includes entities, such as stack or heap pointers, that are often treated as implicit operands. Further, the single assignment requirement applies to all operands and results.

4.2 Redundancy Elimination within a Basic Block

Strictly speaking, our approach aims to eliminate redundant instructions rather than common subexpressions. This includes common subexpression elimination as a special case, since a common subexpression manifests itself as a sequence of redundant instructions; however, it also removes certain kinds of redundancies, such as type tests, that might not be considered to be a common subexpression in a traditional compiler. The principle underlying our algorithm is extremely simple and quite obvious: *two (deterministic) instructions that apply the same operator to identical operands will produce the same results.* The determinacy requirement, which says that the instructions compute functions, is important, but not very restrictive for our application: we do not know of any intermediate representation language for compilers that does not satisfy this requirement. It turns out that by using unification and Prolog meta-level primitives, we are able to exploit this obvious fact in a clean and simple way, obtaining simple and efficient code optimizers without having to worry about any of the low-level clutter associated with common subexpression elimination in traditional compilers.

Assume that the instructions in a basic block are represented as a list of Prolog terms (each instruction is a Prolog term of the form described at the end of the previous section). We assume that we have a set of instructions `Seen` that have already been encountered. The essence of our algorithm is straightforward: given an instruction $I \equiv \mathtt{op}(\mathtt{In}, \mathtt{Out})$ in the basic block, if there is an instruction $I' \equiv \mathtt{op}(\mathtt{In}', \mathtt{Out}')$ in `Seen` such that the operands `In` and `In'` are identical, then

Input : A basic block B of intermediate code instructions.

Output : A modified basic block B with redundant instructions deleted.

Method :

> `Seen` := $\emptyset$;
> **for** each instruction $I \equiv$ `op(In, Out)` in B **do**
>   **if** $\exists$ `op(In′, Out′)` $\in$ `Seen` such that `In == In′` **then**
>   **unify** `Out` and `Out′`;
>   **delete** I from B;
>   **else**
>   **add** I to `Seen`;
>   **fi**
> **od**

Prolog Realization :

```
cse_elim(B_in, B_out) :- cse_elim(B_in, [], B_out).

cse_elim([], S, []).
cse_elim([I1|Rest], Seen, L) :-
     (find(Seen, I2), eqvt(I1, I2)) ->
          cse_elim(Rest, Seen, L)
        ; (L = [I1|Lrest], add(I1, Seen, Seen0),
           cse_elim(Rest, Seen0, Lrest)
          ).

% eqvt(I1, I2) is true iff the instuctions I1 and I2 have
% identical inputs, i.e., are equivalent.  In this case,
% their outputs are unified.

eqvt(I1, I2) :-
     I1 =.. [Op,In1,Out], I2 =.. [Op,In2,Out], In1 == In2.
```

Figure 1: An Algorithm for Redundancy Removal within a Basic Block

their results `Out` and `Out'` must be equal, so we can simply unify `Out` and `Out'` (so that future references to `Out` now also reference `Out'`) and delete I; otherwise, I has not been encountered before, and should be added to `Seen`. The algorithm, and Prolog code realizing it, is given in Figure 1. The efficiency of the Prolog code may be improved by choosing the data structure for `Seen` more carefully, e.g., by indexing it by opcode and passing the opcode of the instruction being considered as a third argument to `find`.

Example 4.1 Consider the following source code statement in a Pascal-like language:

```
a[i, j] := a[i, j] + 1;
```

Assume that the array `a` is stored in row-major order starting at location 1000, that each array element occupies 4 bytes of memory, and that all of its subscripts range over the interval `[1..100]`. Then, the address of `a[i, j]` is given by the following expression (see [1] for details):

$$\begin{aligned} &\mathtt{1000} + \mathtt{4} * \mathtt{100} * (\mathtt{i} - \mathtt{1}) + \mathtt{4}(\mathtt{j} - \mathtt{1}) \\ &= \mathtt{4}(\mathtt{i} * \mathtt{100} + \mathtt{j}) + \mathtt{596}. \end{aligned}$$

Code generated directly, without common subexpression elimination, will repeat this address computation:

```
(1)    mult([100, I], [T1])            /* T1 := 100 * I    */
(2)    add([T1, J], [T2])              /* T2 := T1 + J     */
(3)    mult([4, T2], [T3])             /* T3 := 4 * T2     */
(4)    add([T3, 596], [T4])            /* T4 := T3 + 596   */
(5)    indirect_load([T4], [T5])       /* T5 := *T4        */
(6)    add([T5, 1], [T6])              /* T6 := T5 + 1     */
(7)    mult([100, I], [T7])            /* T7 := 100 * I    */
(8)    add([T7, J], [T8])              /* T8 := T7 + J     */
(9)    mult([4, T8], [T9])             /* T9 := 4 * T8     */
(10)   add([T9, 596], [T10])           /* T10 := T8 + 596 */
(11)   indirect_store([T6], [T10])     /* *T10 := T6       */
```

When our algorithm is executed on this code, no instruction will be eliminated until instruction (7) is processed. Since the inputs to instruction (7) are identical to those of (1), this results in the variables `T7` and `T1` becoming unified and instruction (7) being discarded. The instruction sequence at this point, therefore, is:

```
(1)    mult([100, I], [T1])             /* T1 := 100 * I   */
(2)    add([T1, J], [T2])               /* T2 := T1 + J    */
(3)    mult([4, T2], [T3])              /* T3 := 4 * T2    */
(4)    add([T3, 596], [T4])             /* T4 := T3 + 596  */
(5)    indirect_load([T4], [T5])        /* T5 := *T4       */
(6)    add([T5, 1], [T6])               /* T6 := T5 + 1    */
(8)    add([T1, J], [T8])               /* T8 := T1 + J    */
(9)    mult([4, T8], [T9])              /* T9 := 4 * T8    */
(10)   add([T9, 596], [T10])            /* T10 := T8 + 596 */
(11)   indirect_store([T6], [T10])      /* *T10 := T6      */
```

Notice now that as a result of the unification of `T1` and `T7` at the previous step, the inputs to instructions (2) and (8) become identical. At the next step, therefore, the variables `T2` and `T8` will become unified and instruction (8) will be discarded. This process continues, and the code finally generated does not repeat any of the address computation:

```
(1)    mult([100, I], [T1])             /* T1 := 100 * I   */
(2)    add([T1, J], [T2])               /* T2 := T1 + J    */
(3)    mult([4, T2], [T3])              /* T3 := 4 * T2    */
(4)    add([T3, 596], [T4])             /* T4 := T3 + 596  */
(5)    indirect_load([T4], [T5])        /* T5 := *T4       */
(6)    add([T5, 1], [T6])               /* T6 := T5 + 1    */
(11)   indirect_store([T6], [T4])       /* *T4 := T6       */
```

□

The algorithm, as described above, has two minor shortcomings:

1. It may sometimes fail to detect common subexpressions involving copy statements, i.e., assignments of the form

   `x := y.`

   This is illustrated by the following example: consider the instruction sequence

   ```
   store([1], [X])                      /* X := 1     */
   add([X, Y], [Z])                     /* Z := X + Y */
   add([1, Y], [U])                     /* U := 1 + Y */
   ```

   In this case, the algorithm fails to infer that `1 + Y` in the instruction `add([1, Y], [U])` is a common subexpression. This problem can be taken care of by carrying out *copy propagation* [1] before common subexpression elimination. *A point to note here is that join definitions, i.e. assignments to a variable introduced at join birthpoints during the transformation to static single assignment form, should not be considered during copy propagation, since otherwise the resulting program may no longer be in static single assignment form.*

2. The algorithm does not know about algebraic properties of operations, e.g. that addition is commutative. As a result, it may sometimes fail to detect some common subexpressions. This is illustrated by the following example:

   ```
   add([1, Y], [Z])                       /* Z := 1 + Y */
   add([Z, 2], [X])                       /* X := Z + 2 */
   add([Y, 1], [U])                       /* U := Y + 1 */
   mult([U, 4], [V])                      /* V := U * 4 */
   ```

   In this case, the algorithm fails to infer that `1 + Y` in the instruction `add([Y, 1], [U])` is a common subexpression. This problem can be taken care of by augmenting the Prolog code to express the desired algebraic properties, e.g. by adding clauses of the form

   ```
   eqvt(I1, I2) :-
        1 =.. [Op, [X1,Y1], Out], I2 =.. [Op, [X2,Y2], Out],
        commutative(Op), X1 == Y2, X2 == Y1.

   commutative(add).
   commutative(mult).
   ```

   Even though this is somewhat more complicated than the original definition given in Figure 1, notice that all that we are doing is elaborating, in a clean and logical way, the notion of "equivalence" between two instructions. The point is that the overall algorithm—and anything that depends on it—is not affected, all we are doing is refining the `eqvt/2` relation. Obviously, additional properties could be expressed by suitably elaborating the definition of `eqvt/2`, without affecting any of the remainder of the algorithm. In our experience, this ability to specify aspects of the instruction set in a clean and declarative way is very helpful for verification and modification of the low-level code optimizer (its simplicity, modularity, declarative reading, and ease of modification contrast very pleasantly with the corresponding code that is typically found in traditional compilers).

4.3 Redundancy Elimination across Basic Block Boundaries

Recall that we are considering only loop-free program fragments. It therefore suffices to consider two cases: (i) a fork point, i.e., where a basic block has more than one successor; and (ii) a join point, i.e., where a basic block has more than one predecessor.

Dealing with fork points is straightforward: if a block B has n successors $B_1, \ldots, B_n$ then the initial `Seen` set at the entry to each of the blocks $B_1, \ldots, B_n$ is the `Seen` set at the exit from block B.

For join points, we have to ensure that instructions encountered along one branch leading upto the join point, but not along another branch, are not considered to have been seen when the basic block at the join point is considered. This is easy to handle: consider a basic block B with k predecessors $B_1, \ldots, B_k$, and let the set of instructions seen at the end of a predecessor B_i be $\texttt{Seen}_i$, $1 \leq i \leq k$. Then, the set `Seen` at the entry to B is given by

$$\texttt{Seen} = \bigcap_{i=1}^{k} \texttt{Seen}_i.$$

With this change, the algorithm can be used for common subexpression elimination in any loop-free program.

5 Common Subexpression Elimination in the WAM

The kind of common subexpression most commonly encountered in Prolog programs involves redundant construction of terms [2]. For example, on most Prolog implementations, the clause

```
p([f(X,Y)|L]) :- q(f(X,Y)), p(L).
```

will create two copies of the term `f(X, Y)` each time around the recursion when executed. Since most high-performance Prolog systems are based on the WAM [12], it would be nice if we could adapt our scheme to the WAM. This cannot be done directly, for the following reasons:

1. WAM instructions use implicit arguments, and as a result are context sensitive. For example, a `get_list` or `get_structure` instruction has the registers `S` (the structure pointer) and `H` (the heap pointer), as well the mode bit, as implicit outputs. This is not really a significant problem,

since it can be rectified by rewriting the instructions to make all operands explicit.

2. WAM entities are not single assignment. For example, the various registers can be destructively updated. Again, this does not seem to be a significant problem, since in principle we could consider a transformation to static single assignment form, similar to that discussed earlier for imperative programs, before applying our algorithm.

Unfortunately, as far as we can see, our scheme remains inapplicable to reclaiming common substructures even after these problems have been addressed. There are two (related) reasons for this:

1. Consider a clause '`p :- q(f(a), f(a))`'. The WAM code for this, after we have transformed the instruction set to make all operands explicit, and further have transformed the resulting code to static single assignment form, would be the following, where $\mathtt{S}_i$ and $\mathtt{H}_i$ denote different values of the `S` and `H` registers respectively:

```
p/0 :  put_structure([f/1, H0], [A1, H1, S1])
       unify_constant([a, H1, S1, write_mode], [H2, S2])
       put_structure([f/1, H2], [A2, H3, S3])
       unify_constant([a, H3, S3, write_mode], [H4, S4])
       execute q/2
```

   The two `put_structure` instructions cannot be identified to be equivalent because the register `H`, which points to the top of the heap and is an input to this instruction, is different for the two instructions. As a result, the two occurrences of the common substructure `f(a)` cannot be identified and optimized.

2. Suppose, to get around this problem with failing to identify the common substructure `f(a)`, we ignore the value of the `H` register when determining whether two `put_structure` instructions are equivalent. In this case, things become even worse: consider the clause

```
p :- q(f(a), f(b)).
```

   The code for this is

```
p/0 :  put_structure([f/1, H0], [A1, H1, S1])
       unify_constant([a, H1, S1, write_mode], [H2, S2])
       put_structure([f/1, H2], [A2, H3, S3])
       unify_constant([b, H3, S3, write_mode], [H4, S4])
       execute q/2
```

By ignoring the different values of the H register in the two put_structure instructions, we would erroneously infer that these two instructions are equivalent, and optimize away the second. This would yield the following code:

```
p/0 :  put_structure([f/1, H0], [A1, H1, S1])
       unify_constant([a, H1, S1, write_mode], [H2, S2])
       unify_constant([b, H1, S1, write_mode], [H4, S4])
       execute q/2
```

This is clearly incorrect: register A2 is not set at all any more, and the constant a written onto the heap by the second instruction is immediately overwritten by the constant b as a result of executing the third instruction.

6 Conclusions

Common subexpression elimination is an important low-level compiler optimization. Traditional approaches to implementing this optimization, typically via value numbering schemes, are complicated by a large amount of low-level clutter involved with the maintenance of information about value numbers. In this paper we propose a much simpler scheme for this transformation, using Prolog meta-language features and unification. The scheme is simple and easy to understand, and easy to implement, modify, and extend, and efficient in practice. It considers loop-free programs and assumes that variables are single-assignment entities: thus, if applied to imperative language programs, it requires that they be transformed to static single assignment form. However, since the static single assignment form is useful for a variety of other optimizations, this need not be a great burden. Our scheme is also very flexible in the sense that, unlike common subexpression elimination in most traditional compilers, it is applicable not only to arithmetic, but also to instructions that test types, bounds, etc.

Acknowledgements: Comments by Mats Carlsson and Jan Komorowski were very helpful in improving the contents of the paper. This work was supported in part by the National Science Foundation under grant number CCR-8901283.

References

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

[2] M. Carlsson, personal communication, Jan. 1992.

[3] J. Cocke and J. T. Schwartz, *Programming Languages and their Compilers: Preliminary Notes, Second Revised Version*, Courant Institute of Mathematical Science, New York, 1970.

[4] R. Cytron and J. Ferrante, "What's in a Name? or, The Value of Renaming for Parallelism Detection and Storage Allocation", *Proc. 1987 International Conference on Parallel Processing*, St. Charles, IL, Aug. 1987.

[5] R. Cytron, J. Ferrante, B. Rosen, and M. Wegman, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *ACM Transactions on Programming Languages and Systems* vol. 13 no. 4, pp. 451–490.

[6] S. K. Debray, "QD-Janus: A Prolog Implementation of Janus", research report, Dept. of Computer Science, The University of Arizona, Tucson, May 1991.

[7] A. Houri and E. Shapiro, "A Sequential Abstract Machine for Flat Concurrent Prolog", in *Concurrent Prolog: Collected Papers*, vol. 2, ed. E. Shapiro, pp. 513-574. MIT Press, 1987.

[8] H. J. Komorowski, "Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog", *Proc. Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, Jan. 1982.

[9] J. H. Reif and H. R. Lewis, "Symbolic Evaluation and the Global Value Graph", *Proc. Fourth ACM Symp. on Principles of Programming Languages*, Jan. 1977, pp. 104-118.

[10] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global Value Numbers and Redundant Computations", *Proc. 1988 ACM Symp. on Principles of Programming Languages*, San Diego, CA, Jan. 1988, pp. 12-27.

[11] V. A. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conference on Logic Programming,* Austin, TX, Oct. 1990, pp. 431-446. MIT Press.

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

[13] M. N. Wegman and F. K. Zadeck, "Constant Propagation with Conditional Branches", *ACM Transactions on Programming Languages and Systems* vol. 13 no. 2, April 1991, pp. 181-210.

[14] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.