Automatic Validation of Code-Improving Transformations

Robert van Engelen, David Whalley, and Xin Yuan
Dept. Of Computer Science, Florida State University, Tallahassee, FL 32306

Abstract

Programmers of embedded systems often develop software in assembly code due to inadequate support from compilers and the need to meet critical speed and/or space constraints. Many embedded applications are being used as a component of an increasing number of critical systems. While achieving high performance for these systems is important, ensuring that these systems execute correctly is vital. One portion of this process is to ensure that code-improving transformations applied to a program will not change the program's semantic behavior, which is jeopardized when transformations are specified manually. This paper describes a general approach for validation of many low-level code-improving transformations made either by a compiler or specified by hand. Initially, we associate a region of the program representation with a code-improving transformation. Afterwards, we calculate the region's effects on the rest of the program before and after the transformation. The transformation is considered valid when the effects before and after the transformation are identical.

1 Introduction

Software is being used as a component of an increasing number of critical systems. Ensuring that these systems execute correctly is vital. One portion of this process is to ensure that the compiler produces machine code that accurately represents the algorithms specified at the source code level. This is a formidable task since an optimizing compiler not only translates the source code to machine code, it may apply hundreds or thousands of compiler optimizations to even a relatively small program. However, it is crucial to try to get software correct for many systems. This problem is exacerbated for embedded systems development, where applications are often either developed in assembly code manually or compiler generated assembly is modified by hand to meet speed and/or space constraints. Code-improving transformations accomplished manually are much more suspect than code generated automatically by a compiler.

An optimizing compiler or assembly programmer applies a sequence of code-improving transformations to the representation of a program. Each low-level code-improving transformation consists of a set of changes, where these changes may result in machine instructions being deleted, inserted, or modified. However, the program representation before and after the changes associated with a code-improving transformation should be semantically equivalent.

We use two techniques that dramatically simplify the task of validating code-improving optimizations. First, we show the equivalence of the program representation before and after each improving transformation rather than attempting to show equivalence between the source and object programs directly. While many code-improving transformations may be applied to a program representation, each individual transformation typically consists of only a few changes. Also, if there is an error, then the compiler writer or assembly programmer would find it desirable for a system to identify the transformation that introduced the error. Second, for each code-improving transformation we only attempt to show the equivalence of the region of the program associated with the changes rather than showing the equivalence of the entire program representation. We have found that the region of the program representation that is changed by a code-improving transformation is typically quite small. We show equivalence of the region before and after the transformation by demonstrating that the effects the region will have on the rest of the program will remain the same. In the subsequent sections of the paper we describe the general approach used for validating intraprocedural (within a single function) low-level code-improving transformations and the current status of its implementation.

2 Related Work

There has been much work in the area of attempting to prove the correctness of compilers [5, 6, 8, 9, 14]. Unfortunately, compilers are quite complex programs and proving the correctness of any large program is a difficult task. More success has been made in the area of validating compilations rather than the compiler itself. For instance, equivalence of source and target programs have been verified for an expression language that contains no loops or function calls [4]. Likewise, there has been progress in proving type, memory safeness, and other related properties of a compilation rather than the equivalence of source and target programs [13, 10, 11]. Proving these types of properties is important and these techniques could be used in conjunction with the approach described in this paper, which attempts to prove that the effects a region has on registers and memory are identical before and after a transformation.

Horwitz attempted to identify semantic differences between source programs in a simple high-level language containing a limited number of constructs [7]. First, programs in this language were represented in a *program representation graph* that identifies dependences between statements. Next, a matching function examined both representations to determine if *old* and *new* representations were equivalent. While it is possible that a similar approach using a matching function could be applied on a low-level representation to validate some types of transformations (e.g. ones that change the order of independent instructions such as instruction scheduling), it is unclear how other transformations (e.g. ones that change the form of instructions such as strength reduction) could be validated using this approach. A related approach for proving semantic equivalence of different source programs is to derive normal forms [3]. However, this has only been attempted on a restricted high-level language without loops and function calls.

The *credible compilation* approach [12] attempts to validate code-improving transformations on an intermediate machine-independent representation, but uses a very different approach from the one described in this paper. The compiler writer determines the appropriate type of invariants for the analysis and transformation of each different type of code-improving optimization and the compiler automatically constructs a proof for these invariants for each optimization. While this approach is quite powerful, it puts a burden on a compiler writer to correctly identify the types of changes associated with each optimization and to specify the appropriate invariants that need to be proven correct.

3 Determining the Region of Code Associated with a Transformation

We validate code-improving transformations in the *vpo* compiler [2], which uses RTLs (register transfer lists) to represent machine instructions. The same intermediate representation could be used to validate hand-specified transformations as well. Each register transfer is an assignment that represents a single effect on a register or memory cell of the machine. Thus, the

```
add_pred_blocks_to_region(region, block)
{
   FOR each predecessor P of block DO
      IF (!in_region(region, block)) THEN
         add_block_to_region(P, region);
         add pred_blocks_to_region(P, region);
      block->start = first RTL in block;
      P->end = last RTL in block;
}
```

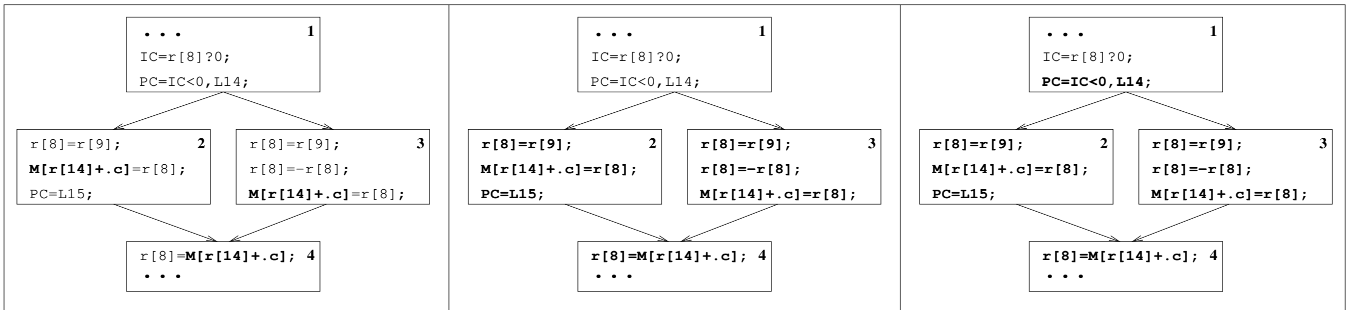
```
calculate_insts_in_region(blkschanged, blksadded,
                          domblk, region)
{
   FOR each block B in blkschanged DO
      B->start = first RTL that was changed in B;
      B->end = last RTL that was changed in B;
      add_block_to_region(B, region);
   FOR each block B in blksadded DO
      B->start = first RTL in B;
      B->end = last RTL in B;
      add_block_to_region(B, region);
   IF (!in_region(region, domblk)) THEN
      add_block_to_region(domblk, region);
   FOR each block B in blkschanged or blksadded DO
      add_pred_blocks_to_region(B, region);
   IF domblk->start == NULL THEN
      domblk->start = conditional branch in domblk;
}
```

Figure 1. Calculating the Extent of a Region

RTL representation served as a good starting point for calculating the semantic effects of a region.

Determining the region of code that is associated with a code-improving transformation requires capturing the changes to the program representation caused by the transformation. We automatically detect changes associated with a transformation by making a copy of the program representation before each code-improving transformation and comparing the program representation after the transformation with the copy. After identifying all of the basic blocks that have been syntactically modified, we find the closest block in the control-flow graph that dominates all of the modified blocks. This dominating block contains the entry point of the region. The region consists of all instructions between the dominating block and the RTLs that have been modified. The region before the transformation is the *old region* and the region after the transformation is the *new region*. These two regions are considered counterparts of each other since they should have the same effects on the rest of the program. Thus, proving program equivalence can be accomplished by proving region equivalence. The effects of the old and new regions are considered semantically equivalent if they are identical at each exit point of the region. Note that the old and new regions need not have the same basic block structure. Only the dominating block and exit points of the two regions have to be identical, which sometimes requires extending one or both of the regions.

Figure 1 shows the algorithm to determine the set of instructions that comprise the region. One should note that a *changed RTL* includes an inserted RTL, an RTL deleted in the counterpart region, or a marker for the point where an RTL was inserted in the counterpart region. The example shown in Figure 2 illustrates how the algorithm works. Consider the program representation shown in Figure 2(a) that depicts the state of the program before a register allocation transformation. The references to variable `c` are to be replaced with a register and these references have been identified and are shown in boldface. The block that most closely dominates all blocks containing the modifications (blocks 2, 3, and 4) is block 1. The region consists of all RTLs between the ones that are changed and this dominating point, which are shown in boldface in Figure 2(b). Block 1 contains no RTLs that have been modified. As shown in Figure 2(c), its conditional branch is included in the region so conditions can be represented when transitions are made to blocks 2 and 3.



(a) Program Representation before a Register Allocation Transformation with Changes Identified

(b) Extent of Region after Propagating to Dominating Block

(c) Extent of Region after Including Branch in Dominating Block

Figure 2. Example of Calculating the Extent of a Region

```
. . .
r[16]=0;
r[17]=HI[_string];
r[19]=r[17]+LO[_string];
r[17]=r[16]+r[19];
. . .
```

(a) Old Region before Extension

```
. . .
r[16]=0;
r[17]=HI[_string];
r[19]=r[17]+LO[_string];
r[17]=r[19];
. . .
```

(b) New Region before Extension

```
. . .
r[16]=0;
r[17]=HI[_string];
r[19]=r[17]+LO[_string];
r[17]=r[16]+r[19];
. . .
```

(c) Old Region after Extension

```
. . .
r[16]=0;
r[17]=HI[_string];
r[19]=r[17]+LO[_string];
r[17]=r[19];
. . .
```

(d) New Region after Extension

Figure 3. Example of Extending the Scope of a Region

There are cases when the extent of a region has to be recalculated. For instance, the points at which one region exits has to be identical to the exit points in its counterpart region. If an exit point in one region does not exist in its counterpart, then that exit point is added to its counterpart region and the extent of the region is recalculated. Sometimes a region needs to be extended since its effects are not the same as its counterpart region. For instance, consider Figures 3(a) and 3(b). Only one change was detected, so the old and new regions initially consist of a single instruction shown in boldface. Obviously, these two regions in isolation do not have the same effect. However, there is a reference to `r[16]` in one region that is not in the other. If the effects of the two regions are not identical and there are more uses or sets of a specific register or a variable in one region, then the regions are extended to include an additional set of that register or variable. Figures 3(c) and 3(d) show the extension of the old and new regions to include the set of `r[16]`, which allows identical effects to be calculated for each region.

4 Calculating the Effects of a Region

Each region consists of a single entry point and one or more exit points. A separate set of effects is calculated for each exit point from the region. The old and new regions are considered equivalent only if for each exit point they have the same effects. Figure 4 shows the algorithm to calculate the effects of a region. Loops are processed innermost first. The effects of each node at the same loop level are calculated only after all of its non-back edge predecessors have been processed. More details regarding this algorithm will be illustrated in the following subsections.

```
merge_effects_in_node(n, effects)
{
   effects = "";
   FOR each RTL r in n DO
      merge_effects(effects, r);
}


calc_pred_effects(preds, preds_effects, guard)
{
   p = first pred in preds;
   guard = p->guard;
   FOR each remaining p in preds DO
      guard ||= "∨" || p->guard;
   FOR each unique dst d
       of the effects in preds DO
      new_effect = "<d>=";
      FOR each effect e in preds DO
         IF e->dst == d THEN
            new_effect ||=
               "<e->src> if <guard of e>";
      IF other preds q that did not set d THEN
         new_effect ||=
            "<d> if (∨ of all guards of q)";
      new_effect ||= ");";
}
```

```
process_node_in_region(n)
{
   IF any unprocessed pred of n THEN
      return;
   calc_pred_effects(n->preds, pred_effects, n->guard);
   merge_effects(preds_effects, n->effects);
   add_guard_to_effects(n->guard, n->effects);
   mark n as being processed;
   FOR each successor s of n DO
      process_node_in_region(s);
}

calculate_region_effects(region)
{
   FOR each node n in region DO
      merge_effects_in_node(n);
   FOR each loop l in region (innermost first) DO
      process_node_in_region(l->header->node);
      calculate_exit_condition
         (l->effects, exit_cond);
      replace_recurrs_with_fp_funcs
         (l->effects, exit_cond);
      replace_loop_with_single_node
         (region, l->effects);
   process_node_in_region(region->top);
}
```

Figure 4. Calculating the Effects of a Region

4.1 Merging Effects

Effects in a region are merged for two reasons. First, merging obtains an order-independent representation of effects. Second, merging eliminates the use of temporaries within the region. Figure 5 displays the merging of effects in the regions shown in Figure 3(c) and Figure 3(d). Each RTL is merged into the effects one at a time. Note that when the destination of an effect is no longer live, then the effect is deleted. The point where a register dies is depicted to the right of that RTL. The liveness of registers and variables in a region was calculated using a demand-driven approach, as opposed to the traditional dataflow analysis used in *vpo*. For instance, step 2 in Figure 5(a) deletes the effect that updates `r[17]` since the register is no longer live. The final effects of the old and new regions in Figure 5 will be identical after simplification, which is described later in the paper.

The actual implementation of the merging was accomplished using a directed acyclic graph (DAG), which effectively conserves storage since common-subexpressions occurring in the effects of a region are stored only once. A DAG node is created for each source term in an RTL assignment and the source term is replaced by a reference to the node. For example, the merging of `r[10]=r[8]-2;` with `r[11]=r[10]+r[10];` results in `r[10]=`n_1 and `r[11]=`n_2, where two DAG nodes n_1 and n_2 are created with n_1=`r[8]-2` and n_2=n_1+n_1. Without a DAG representation for effects, merging can result in exponentially growing storage requirements for effects.

4.2 Representing Conditional Effects

Sometimes assignments of expressions to registers or variables need to be conditional. Conditional assignments can occur due to two reasons. First, *vpo* cannot always determine if the addresses of two memory references are equal. Consider

```
0. r[16]=0;
   r[17]=HI[_s];
   r[19]=r[17]+LO[_s];    r[17]:
   r[17]=r[16]+r[19];     r[16]:

1. r[17]=HI[_s]; r[16]=0;
   r[19]=r[17]+LO[_s];    r[17]:
   r[17]=r[16]+r[19];     r[16]:

2. r[19]=HI[_s]+LO[_s]; r[16]=0;
   r[17]=r[16]+r[19];     r[16]:

3. r[17]=0+HI[_s]+LO[_s]; r[19]=HI[_s]+LO[_s];
```

(a) Merging Effects in Old Region

```
0. r[16]=0;                r[16]:
   r[17]=HI[_s];
   r[19]=r[17]+LO[_s];     r[17]:
   r[17]=r[19];

1. r[17]=HI[_s];
   r[19]=r[17]+LO[_s];     r[17]:
   r[17]=r[19];

2. r[19]=HI[_s]+LO[_s];
   r[17]=r[19];

3. r[17]=HI[_s]+LO[_s]; r[19]=HI[_s]+LO[_s];
```

(b) Merging Effects in New Region

Figure 5. Merging Effects within a Single Block

```
M[r[2]]=r[4];
r[5]=M[r[3]];
```

(a) Potential Set/Use Alias Region before Merging

$$\texttt{r[5]=}\left\{\begin{array}{ll}\texttt{r[4]} & if\ \texttt{r[2]} = \texttt{r[3]} \\ \texttt{M[r[3]]} & if\ \texttt{r[2]} \neq \texttt{r[3]}\end{array}\right\}\texttt{; M[r[2]]=r[4];}$$

(b) Potential Set/Use Alias Region after Merging

```
M[r[2]]=r[4];
M[r[3]]=r[5];
```

(c) Potential Set/Set Alias Region before Merging

$$\texttt{M[r[3]]=r[5]; M[r[2]]=}\left\{\begin{array}{ll}\texttt{r[5]} & if\ \texttt{r[2]} = \texttt{r[3]} \\ \texttt{r[4]} & if\ \texttt{r[2]} \neq \texttt{r[3]}\end{array}\right\}\texttt{;}$$

(d) Potential Set/Set Alias Region after Merging

Figure 6. Conditional Effects Due to Potential Aliasing

merging the effects for the region in Figure 6(a), where a set and a use may reference the same memory location. The value assigned to `r[5]` depends on whether `r[2]` equals `r[3]`. Figure 6(b) shows the merged effect with a guarded expression. All guards in a guarded expression are disjoint. Figures 6(c) and 6(d) show how guards are introduced due to potential aliasing from two sets to memory. An alias between a use followed by a set does not require a conditional assignment when merged since the set cannot affect that use. Note that aliases due to loop carried dependences will be handled by our representation of effects in loops, which will be described later.

Conditional assignments also occur due to conditional control flow. Consider the region in Figure 7. Step 1 shows the results after merging effects in each of the three basic blocks separately. Note that the value assigned to the PC (program counter) is a guarded expression. Step 2 shows the effects from block 1 on the transition to block 2. A guarded expression is now assigned to the variable `val` since a change in state should only occur in this effect if the transition to block 2 is taken. Step 3 shows comparable effects from block 1 on the transition to block 3. Step 4 shows the effects in block 2 after merging with the effects in block 1. The effect from block 1 shown in step 2 is removed since we have an assignment to the same variable in block 2. Guards are propagated along transitions between nodes in the region. Thus, the assignment in block 2 only occurs when `r[8]<0`. Step 5 shows the effect in block 3 after merging with the effects in blocks 1 and 2.

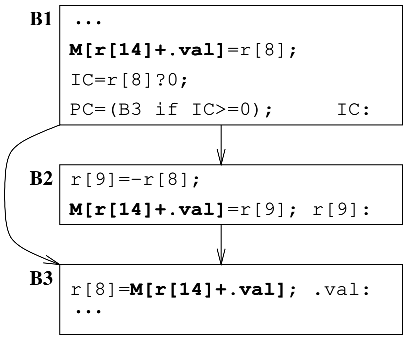
1. Block 1 after merging effects:
   `M[r[14]+.val]=r[8]; PC=(B3` *if* `r[8]≥0);`
   Block 2 after merging effects:
   `M[r[14]+.val]=-r[8];`
   Block 3:
   `r[8]=M[r[14]+.val];`

2. Effects from block 1 on transition to block 2:
   `M[r[14]+.val]=(r[8]` *if* `r[8]<0);`

3. Effects from block 1 on transition to block 3:
   `M[r[14]+.val]=(r[8]` *if* `r[8]≥0);`

4. Block 2 after merging effects with block 1:
   `M[r[14]+.val]=(-r[8]` *if* `r[8]<0);`

5. Block 3 after merging effects with blocks 1 and 2:
   $$\texttt{r[8]}=\left\{\begin{array}{ll}\texttt{r[8]} & if\ \texttt{r[8]}\geq 0\\ \texttt{-r[8]} & if\ \texttt{r[8]}<0\end{array}\right\}\ if\ \texttt{r[8]}\geq 0 \vee \texttt{r[8]}<0;$$



6. Block 3 after simplification:
   $$\texttt{r[8]}=\left\{\begin{array}{ll}\texttt{r[8]} & if\ \texttt{r[8]}\geq 0\\ \texttt{-r[8]} & if\ \texttt{r[8]}<0\end{array}\right\};$$

Figure 7. Example of Calculating Effects with Conditional Control Flow

4.3 Representing Effects from Loops

A region may span multiple basic blocks that cross loop nesting levels. Merging the effects across loop nesting levels requires calculating the effects of an entire loop. One issue that must be addressed is how to represent a recurrence, which involves the use of a variable or register that is set on a previous loop iteration. An induction variable is one example of a simple recurrence. We represent a recurrence as a recursive function using the following notation. The *label* distinguishes the recurrence from other recurrences. The *new value* represents the next value of the recurrence. References to *w* in the *new value* represent the previous value of the recurrence. The *initial value* is the initial value of the recurrence. The *condition* indicates when the recurrence is no longer applied. Thus, this notation is used to represent a sequential ordering of effects, where each instance represents the effect on a different iteration of a loop.

$$y(\langle label\rangle, \langle new\ value\rangle, \langle initial\ value\rangle)\ until\ \langle condition\rangle$$

This notation has been chosen to roughly resemble the application of the fixed-point **Y** combinator of combinatory calculus:

$$\mathbf{Y}\ (\lambda \texttt{w}.\text{if } \langle condition\rangle \text{ then } \langle new\ value\rangle \text{ else } \texttt{w})\ \langle init\ value\rangle$$

A *sequence* can be used to represent recursive functions when the *new value* is obtained by incrementing the current value, which is the case for basic induction variables. We adopt a notation that is similar to the notation used for chains of recurrences [1, 16] (CRs). CRs represent Newton series conversion for polynomials. Each sequence has the following form, which is similar to a recursive function. Unlike recursive functions, some arithmetic operations can be applied to sequences.

$$\{\langle label\rangle, \langle initial\ value\rangle, \langle increment\rangle\}\ until\ \langle condition\rangle$$

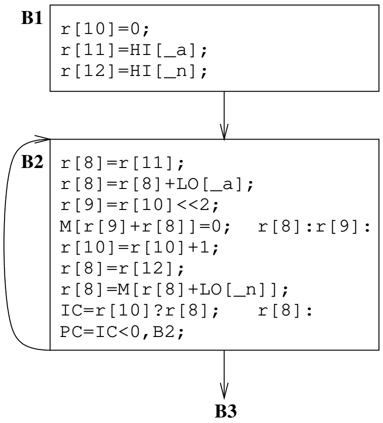
1. Block 1 after merging effects within the block:
   r[10]=0; r[11]=HI[_a]; r[12]=HI[_n];
   Block 2 after merging effects within the block:
   M[r[10]<<2+r[11]+LO[_a]]=0; r[10]=r[10]+1;

2. Block 1:
   r[10]=0; r[11]=HI[_a]; r[12]=HI[_n];
   Block 2 introducing a recursive function for recurrences:
   M[(y(B2,w+1,r[10]) *until*
   y(B2,w+1,r[10]+1)≥M[r[12]+LO[_n]])<<2+r[11]+LO[_a]]=0;
   r[10]=y(B2,w+1,r[10]) *until*
   y(B2,w+1,r[10]+1)≥M[r[12]+LO[_n]];

3. After merging the first block into the loop:
   M[(y(B2,w+1,0) *until*
   y(B2,w+1,0+1)≥M[HI[_n]+LO[_n]])<<2+HI[_a]+LO[_a]]=0;
   r[10]=y(B2,w+1,0) *until* y(B2,w+1,0+1)≥M[HI[_n]+LO[_n]];

4. After simplification:
   M[(y(B2,w+1,0) *until* y(B2,w+1,1)≥M[_n])*4+_a]=0;
   r[10]=y(B2,w+1,0) *until* y(B2,w+1,1)≥M[_n];

5. After changing the recursive functions to sequences:
   M[({B2,0,1} *until* y(B2,w+1,1)≥M[_n])*4+_a]=0;
   r[10]={B2,0,1} *until* y(B2,w+1,1)≥M[_n];



6. After adjusting initial value and increment of the first sequence:
   M[({B2,_a,4} *until* y(B2,w+1,1)≥M[_n])]=0;
   r[10]={B2,0,1} *until* y(B2,w+1,1)≥M[_n];

7. After detecting that r[10] is not used after the loop:
   M[({B2,_a,4} *until* y(B2,w+1,1)≥M[_n])]=0;

Figure 8. Example of Calculating Effects for a Loop

Figure 8 depicts an example of representing the effects of a loop. In this example, the loop is about to be modified by a loop strength reduction transformation. Step 1 shows the effects of the preheader and loop blocks after merging the effects of the blocks separately. Step 2 shows the effects of the loop block after replacing all uses of r[10] with a recursive function since the assignment (r[10]=r[10]+1;) is a recurrence. Note that the *condition* of the recursive function is the exit condition of the loop. Step 3 shows the effects after merging the preheader block with the loop block. The register r[10], unlike registers r[11] and r[12], is not loop invariant. However, the only explicit uses of r[10] in the loop after applying the recursive functions is in the *initial value* of these functions. Thus, assignments before a loop can be successfully merged into a loop after representing recurrences with recursive functions. Likewise, effects after a loop will be guarded by the exit condition, which will allow merging of effects in a loop with effects after the loop. Step 4 shows the effects after simplification. Constant expressions are simplified, the sum of the high and low portions of a value is replaced with the value itself, and left shifts by a constant are replaced by multiplies. Step 5 changes the recursive functions to sequences. Step 6 adjusts the first sequence by a factor and an offset. Finally, step 7 deletes the assignment to r[10] since r[10] is not used after the loop. Recurrences can be viewed as unrolling the iterations of a loop until the condition is satisfied. Multiple recurrences refering to the same loop are unrolled in parallel. For instance, the effect of step 7 in Figure 8 can be viewed as follows when the value of M[_n] is assumed to be 10.

$$\begin{aligned}
\mathtt{M[_a]} &= (\mathtt{0}\ \mathit{if}\ \mathtt{1} < \mathtt{M[_n]}); \\
\mathtt{M[_a+4]} &= (\mathtt{0}\ \mathit{if}\ \mathtt{2} < \mathtt{M[_n]}); \\
&\vdots \\
\mathtt{M[_a+36)]} &= (\mathtt{0}\ \mathit{if}\ \mathtt{10} < \mathtt{M[_n]});
\end{aligned}$$

5 Simplifying Effects

Merging and subsequent simplification of effects yields canonical representations allowing structural comparison between effects to prove that they are identical. The canonical representations of the effects corresponding to the exit points of old and new regions are compared by *vpo* to determine that the semantic effect of the transformed region of code is unchanged.

A normal form defines an equivalence class for all the semantic effects that are identical. Normal forms are essential for theorem proving systems that attempt to prove the equivalence between two programs, see e.g. [3]. However, in our approach the programs compared are not arbitrary but have many parts in common, because the difference is the application of a single transformation on a program. This greatly alleviates the necessity of deriving normal forms for effects, because it suffices to simplify effects by rules that apply partial evaluation on RTLs and rules that can normalize arithmetic, conditional, and logical expressions. It is known that normal forms do not exist for logical expressions involving relational operators on numeric values. These types of expressions frequently occur in effects (e.g. as guards). This means that the equivalence of the modified region is a sufficient condition for the correctness of a transformation, but it not a necessary condition.

We developed a rewrite system that derives normal forms whenever possible (e.g. expanded forms for arithmetic expressions and disjunctive normal forms for logical expressions) but does not attempt to calculate normal forms for logical expressions involving relations. Figure 9 lists a subset of the rules that were used to simplify effects. Since `VPO` will not use transitivity of equalities and inequalities in code improving transformations, the normalization of logical expressions does not have to exploit transitivity in the simplification. The simplification rules are fixed, and the rules do not have to be extended to validate new code improving transformations, because the canonical representations are derived by partial evaluation of the instructions of the affected regions of the transformation.

Un-nesting of guards (rules 10 and 11) flattens the guarded expressions, thereby removing any specific evaluation order of conditions and enabling the removal of redundant conditions by simplification. It is assumed that the guarded expressions have branches whose conditions are disjoint. This is always guaranteed, because the guarded conditions are formed by the conditions on the different paths in the control flow graph, and no two paths can be taken at the same time. As a result, the branches of a guarded expression can be lexicographically ordered to form a canonical representation. Using this reordering property, two branches can be combined (rule 12) when the guarded expressions in the branches are identical.

The replacement of conditional operands of an operator by a condition guarding the application of the operator is called lifting of guards (rules 13–15). Global common-subexpression elimination (CSE) in *vpo* may result in operands that are evaluated under different conditions after the CSE transformation. Lifting ensures that the effects under these conditions are unaffected. The normalization of relations on integers (rules 16–23) ensures that relations such as `r[8]>0` and `r[8]>=1` have the same canonical representation. It also enables the absorption (rules 4–9) to match relational expressions that are inverses.

Each exit point's effects of the region need to be simplified and compared with its counterpart effects within *vpo*. While analyzing the control–flow graph of the affected region of code, *vpo* builds a directed acyclic graph (DAG) representation

#	Rule	Description
	Translation to Disjunctive Normal Form:	
1	$(X \vee Y) \wedge Z \Rightarrow (X \wedge Y) \vee (X \wedge Z)$	distribution
2	$\neg(X \vee Y) \Rightarrow \neg X \wedge \neg Y$	De Morgan
3	$\neg(X \wedge Y) \Rightarrow \neg X \vee \neg Y$	De Morgan
4	$X \vee (X \wedge Y) \Rightarrow X$	absorption
5	$X \vee (\neg X \wedge Y) \Rightarrow X \vee Y$	absorption
6	$\neg X \vee (X \wedge Y) \Rightarrow \neg X \vee Y$	absorption
7	$(X \wedge Y) \vee (\neg X \wedge Y) \Rightarrow Y$	absorption
8	$(X \wedge Y \wedge Z) \vee (\neg X \wedge Z) \Rightarrow (Y \wedge Z) \vee (\neg X \wedge Z)$	absorption
9	$(\neg X \wedge Y \wedge Z) \vee (X \wedge Z) \Rightarrow (Y \wedge Z) \vee (X \wedge Z)$	absorption
	Guarded expression normalization:	
10	$\left\{\begin{array}{ll} X & if\ Y \\ \vdots & if\ \vdots \end{array}\right\} if\ Z \Rightarrow \left\{\begin{array}{ll} X & if\ Y \wedge\ Z \\ \vdots & if\ \vdots \wedge Z \end{array}\right\}$	left un-nesting of guards
11	$X\ if \left\{\begin{array}{ll} Y & if\ Z \\ \vdots & if\ \vdots \end{array}\right\} \Rightarrow \left\{\begin{array}{ll} X & if\ Y \wedge Z \\ X & if\ \vdots \wedge \vdots \end{array}\right\}$	right un-nesting of guards
12	$\left\{\begin{array}{ll} \ldots & \\ X & if\ Y \\ \ldots & \\ X & if\ Z \\ \ldots & \end{array}\right\} \Rightarrow \left\{\begin{array}{ll} \ldots & \\ X & if\ Y \vee Z \\ \ldots & \end{array}\right\}$	combining
	Lifting of guards (where $\oplus$ is a dyadic operator and $\ominus$ is a monadic operator):	
13	$X \oplus \left\{\begin{array}{ll} Y & if\ Z \\ \vdots & if\ \vdots \end{array}\right\} \Rightarrow \left\{\begin{array}{ll} X \oplus Y & if\ Z \\ X \oplus \vdots & if\ \vdots \end{array}\right\}$	
14	$\left\{\begin{array}{ll} X & if\ Y \\ \vdots & if\ \vdots \end{array}\right\} \oplus Z \Rightarrow \left\{\begin{array}{ll} X \oplus Z & if\ Y \\ \vdots \oplus Z & if\ \vdots \end{array}\right\}$	
15	$\ominus \left\{\begin{array}{ll} Y & if\ Z \\ \vdots & if\ \vdots \end{array}\right\} \Rightarrow \left\{\begin{array}{ll} \ominus Y & if\ Z \\ \ominus \vdots & if\ \vdots \end{array}\right\}$	
	Normalization of relations on integers:	
16	$X = Y \Rightarrow X - Y = 0$	when $X - Y <_{\mathrm{lex}} Y - X$ and $Y \neq 0$
17	$X = Y \Rightarrow Y - X = 0$	when $X - Y \geq_{\mathrm{lex}} Y - X$ and $Y \neq 0$
18	$X > Y \Rightarrow X - Y > 0$	when $X - Y <_{\mathrm{lex}} Y - X$ and $Y \neq 0$
19	$X > Y \Rightarrow \neg(Y - X + 1 > 0) \vee \neg(Y - X > 0)$	when $X - Y \geq_{\mathrm{lex}} Y - X$ and $Y \neq 0$
20	$X \geq Y \Rightarrow X - Y + 1 > 0$	when $X - Y <_{\mathrm{lex}} Y - X$ and $Y \neq 0$
21	$X \geq Y \Rightarrow \neg(Y - X > 0)$	when $X - Y \geq_{\mathrm{lex}} Y - X$ and $Y \neq 0$
22	$X \neq Y \Rightarrow \neg(X - Y = 0)$	when $X - Y <_{\mathrm{lex}} Y - X$ and $Y \neq 0$
23	$X \neq Y \Rightarrow \neg(Y - X = 0)$	when $X - Y \geq_{\mathrm{lex}} Y - X$ and $Y \neq 0$
	Relational simplification:	
24	$X > 0 \wedge Y > 0 \Rightarrow X > 0$	when $X \leq Y$
25	$X > 0 \wedge Y > 0 \Rightarrow Y > 0$	when $X > Y$
26	$X > 0 \wedge Y = 0 \Rightarrow false$	when $X \leq Y$
27	$X > 0 \wedge Y = 0 \Rightarrow Y = 0$	when $X > Y$
28	$X > 0 \vee Y > 0 \Rightarrow Y > 0$	when $X \leq Y$
29	$X > 0 \vee Y > 0 \Rightarrow X > 0$	when $X > Y$
30	$X > 0 \vee Y = 0 \Rightarrow X + 1 > 0$	when $X = Y$
31	$X > 0 \vee Y = 0 \Rightarrow X > 0$	when $X > Y$

Figure 9. Subset of the Effect Simplification Rules

for the effects, as described previously. The nodes in the final DAG (obtained after all effects have been merged) are marked when the node is used in an effect corresponding to an exit point of the region of code. The marking proceeds by recursively analyzing the expressions stored at the nodes in the DAG. Marking the DAG effectively eliminates unused expressions. Unused expressions typically are expressions assigned to registers or variables that are not live at the exit point of the region.

For each marked node visited in a postorder traversal of the DAG, the term stored at the marked node is simplified if not already simplified by rewriting the term into a canonical representation, where references in the term to other nodes are replaced by references to canonical representations of the terms in the nodes. When calculating the canonical representation of an effect, embedded references to canonical representations need not be simplified again. Finally, the effects at the exit points of the regions are simplified and the DAG node references are replaced by their canonical representations.

The *vpo* compiler applies thousands of code improving transformations, each of which has to be validated by calculating effects of the before and after regions of code affected by the transformation. It is possible that one could use an existing computer algebra system (e.g. Maple^{tm}, Mathematica^{tm}, Reduce^{tm}) for the purpose of expression simplification. However, this approach would be hampered by implementation problems. To integrate a computer algebra system with another program, a software bus (e.g. MathLink^{tm}) would have to be adopted for exchanging symbolic data between the computer algebra system and the program. The overhead of exchanging symbolic expressions over a software bus is prohibitive. On both sides of the software bus, expressions have to be represented internally and in possibly different formats. The storage saving DAG representation in *vpo* for effects and their canonical representations cannot be adopted when a software bus is used.

5.1 Simplifying Effects Using Ctadel

We modified the *vpo* compiler with calls to a Prolog engine to simplify effects. The Prolog engine executes a version of the CTADEL system [15], which is an extensible rule-based symbolic manipulation program implemented in SWI-Prolog [17]. The expression simplification is applied in the address space of the *vpo* compiler by linking the SWI-Prolog interpreter with *vpo*.

Besides the efficiency consideration of linking Prolog to the *vpo* compiler, another reason for using CTADEL is to be able to easily implement a rule-base for simplifying effects. This allows us to experiment with the effect simplification more easily as compared to implementing the effect simplification directly in C. We implemented numerous rules in CTADEL to simplify effects. A large portion of the rules are for primitive arithmetic operations (comparable to constant folding applied by compilers). Some of the more advanced rules were shown in Figure 9.

The matching mechanism in CTADEL for applying rules is very powerful. The system respects the properties of operators in an expression to align the subexpression with the left-hand side of a transformation rule. Associative and commutative properties of operators are declared and the system will apply the rules by taking these properties into account. For example, rule #8 is simply written in CTADEL as

```
(X/\Y/\Z)\/(\X/\Z) => (Y/\Z)\/(\X/\Z).
```

where the associative and commutative operators `/\` and `\/` denote logical conjunction and disjunction, respectively, and `\`

denotes logical negation. When the rule is matched against the example expression

$$(a = 0 \wedge b = 0 \wedge c = 0 \wedge d = 0) \vee (\neg b = 0 \wedge a = 0 \wedge d = 0)$$

the system determines that $X \leftarrow (b = 0)$, $Y \leftarrow (d = 0)$, and $Z \leftarrow (a = 0 \wedge c = 0)$ and simplifies the expression into $(d = 0 \wedge a = 0 \wedge c = 0) \vee (\neg b = 0 \wedge a = 0 \wedge c = 0)$.

The symbolic simplification of effects is illustrated by an example depicted in Figure 10. The calculation of effects proceeds from the dominating block of the region (**B1**). The guarding conditions that result from conditional control flow are propagated down and used to guard RTLs. Effects from blocks are merged as described in Section 4.

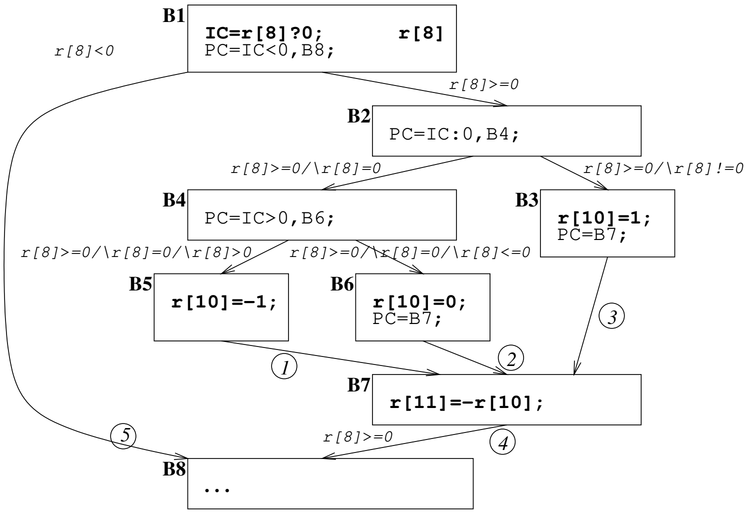
Note that the merging requires extensive symbolic simplification of logical expressions and the application of rules for un-nesting and lifting guarded expressions. Further note that the final effects accurately describe the semantics of the region of code. Clearly, block **B5** is unreachable code. The semantic effect of the code after a dead-code elimination transformation has been applied is unchanged (not shown). Hence, the dead-code elimination transformation is valid.

6 Current Status

We have modified the *vpo* compiler to validate code-improving transformations using the techniques that were described previously in this paper. Table 1 shows some small test programs that we have compiled while validating code-improving transformations. The third column indicates the number of improving transformations that were applied during the compilation of the program. The fourth column represents the percentage of the transformations where the effects of the old and new regions associated with the transformation were identical. The only transformations that have not been validated are those with regions that span basic blocks at different loop nesting levels since the ability to represent effects containing entire loops (as shown in Section 4.3) has not yet been implemented. All transformations applied to these test programs with regions within a single loop level have been validated. We also plan to later test our validation approach with larger test programs as well. The fifth column represents the average static number of instructions for each region associated with all code-improving transformations during the compilation. This average illustrates that the typical region associated with a transformation is quite small. The final column denotes the ratio of compilation times when validating programs versus a normal compilation.

Program	Description	Num Trans	Validated	Region Size	Overhead
ackerman	benchmark that performs recursive function calls	89	100.0%	3.18	13.64
arraymerge	benchmark that merges two sorted arrays	483	89.2%	4.23	63.89
banner	poster generator	385	90.6%	5.42	34.13
bubblesort	benchmark that performs a bubblesort on an array	342	85.4%	6.10	34.37
cal	calendar generator	790	91.1%	5.16	105.64
head	displays the first few lines of files	302	89.4%	8.42	152.64
matmult	multiplies 2 square matrices	312	89.7%	5.55	28.97
puzzle	benchmark that solves a puzzle	1928	78.5%	5.85	128.98
queens	eight queens problem	296	85.8%	6.79	73.65
sieve	finds all prime numbers between 3 and 16383	217	80.6%	6.85	21.90
sum	prints the checksum and block count for a file	235	91.9%	8.62	195.19
uniq	report or filter out repeated lines in a file	519	91.1%	4.21	163.26
average		492	88.6%	5.87	84.64

Table 1. Benchmarks



1. At transition (1) from block **B5** after merging we find $\mathtt{r[10]} = \{-1 \; if \; \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0\}$
2. At transition (2) from block **B6** after merging we find $\mathtt{r[10]} = \{0 \; if \; \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0\}$
3. At transition (3) from block **B3** after merging we find $\mathtt{r[10]} = \{1 \; if \; \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0\}$
4. After combining the effects of the blocks **B5**, **B6**, and **B3** we obtain

$$\mathtt{r[10]} = \left\{ \begin{array}{ll} 1 & if \; \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0 \\ 0 & if \; \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0 \\ -1 & if \; \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} 1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \end{array} \right\}$$

5. Merging the above effects with the effects of block **B7** yields

$$\begin{array}{rcl} \mathtt{r[10]} & = & \left\{ \begin{array}{ll} 1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \end{array} \right\} \\ \mathtt{r[11]} & = & -\left\{ \begin{array}{ll} 1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} -1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \end{array} \right\} \end{array}$$

6. Merging the (empty) effects at transition (5) with the effects at transition (4) we obtain the effects of the region of code

$$\begin{array}{rcl} \mathtt{r[10]} & = & \left\{ \begin{array}{ll} \left\{ \begin{array}{ll} 1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \end{array} \right\} & if \; \mathtt{r[8]} \geq 0 \\ \mathtt{r[10]} & if \; \mathtt{r[8]} < 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} 1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \\ \mathtt{r[10]} & if \; r[8] < 0 \end{array} \right\} \\ \mathtt{r[11]} & = & \left\{ \begin{array}{ll} \left\{ \begin{array}{ll} -1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \end{array} \right\} & if \; \mathtt{r[8]} \geq 0 \\ \mathtt{r[11]} & if \; \mathtt{r[8]} < 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} -1 & if \; \mathtt{r[8]} > 0 \\ 0 & if \; \mathtt{r[8]} = 0 \\ \mathtt{r[11]} & if \; r[8] < 0 \end{array} \right\} \end{array}$$

where the guard condition $\mathtt{r[8]} \geq 0$ is derived by forming the disjunction of the guard conditions on the incoming edges to block **B7**, which is the simplified form of

$$(\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0) \vee (\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0) \vee (\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0)$$

Figure 10. Example of Simplifying Effects

The use of an interpretive Prolog system to simplify effects did impact the speed of the validation process. However, an overhead of about two orders of magnitude would probably be acceptable to most people, as compared to the cost of not detecting potential errors. In addition, validation would not be performed on every compilation. If excessive compilation time becomes an issue, then we will investigate slicing the regions to reduce the number of effects to simplify.

A variety of types of transformations in the *vpo* compiler have been validated using our approach. These transformations include *algebraic simplification of expressions*, *basic block reordering*, *branch chaining*, *common subexpression elimination*, *constant folding*, *constant propagation*, *unreachable code elimination*, *dead store elimination*, *evaluation order determination*, *filling delay slots*, *induction variable removal*, *instruction selection*, *jump minimization*, *register allocation*, *strength reduction*, and *useless jump elimination*. Unlike an approach that requires the compiler writer to provide invariants for each different type of code-improving transformation [12], our general approach was applied to all of these transformations without requiring any special information. Thus, we believe that our approach could be used to validate many hand-specified transformations on assembly code by programmers of embedded systems.

7 Conclusions

This paper has described a general approach for validating low-level code-improving transformations. First, the region in the program representation associated with the changes caused by a code-improving transformation is identified. Second, the effects of the region before and after the transformation are calculated. Third, a set of rules are applied in an attempt to obtain a normal form of these effects. Finally, the effects of the region before and after the transformation are compared. If the two sets of effects are identical, then the transformation is deemed valid. One should note that the approach presented in this paper does not guarantee to show that two *arbitrary* regions are semantically equivalent. However, we have demonstrated that it is feasible to use our approach to validate many conventional code-improving transformations.

Validating code-improving transformations has many potential benefits. Validation provides greater assurance of correct compilation of programs, which is important since software is being used as a component of an increasing number of critical systems. The time spent by compiler writers to detect errors can be dramatically reduced since the transformations that do not preserve the semantics of the program representation are identified during the compilation. Finally, validation of hand-specified transformations on assembly code can be performed, which can assist programmers of embedded systems.

References

[1] O. Bachmann, P.S. Wang, and E.V. Zima. Chains of recurrences - a method to expedite the evaluation of closed -form functions. In *International Symposium on Symbolic and Algebraic Computing*, pages 242–249, Oxford, 1994. ACM.

[2] M. E. Benitez and J. W. Davidson. A Portable Global Optimizer and Linker. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

[3] J.A. Bergstra, T.B. Dinish, J. Field, and J. Heering. A complete transformational toolkit for compilers. In H.R. Nielson, editor, 6th *European Symposium on Programming (ESOP'96), LNCS 1058*, Linköping, Sweden, April 1996. Springer.

[4] A. Cimatti and et. al. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In *International Conference on Computer Aided Verification*, pages 202–213, June 1997.

[5] P. Dybjer. Using Domain Algebras to Prove the Correctness of a Compiler. *Lecture Notes in Computer Science*, 182:329–338, 1986.

[6] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a Verified Implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.

[7] S. Horwitz. Identifying the Semantic and Textual Differences between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[8] J. Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[9] F. Morris. Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 144–152, 1973.

[10] G. Necula. Proof-Carrying Code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[11] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[12] M. Rinard and D. Marinov. Credible Compilation with Pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verfication*, 1999.

[13] D. Tarditi, J. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[14] J. Thatcher, E. Wagner, and J. Wright. More on Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of a Workshop on Semantics-Directed Compiler Generation*, pages 165–188, 1994.

[15] R. van Engelen, L. Wolters, and G. Cats. Ctadel: A generator of multi-platform high performance codes for pde-based scientific applications. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 86–93, May 1996.

[16] R.A. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical report, TR-000102, Computer Science Deptartment, Florida State Universi ty, 2000. Available from http://www.cs.fsu.edu/˜engelen/cr.ps.gz.

[17] J. Wielemaker. *SWI-Prolog Reference Manual*. University of Amsterdam, 1995. Available by anonymous ftp from `swi.psy.uva.nl`.