

# Imperative-Program Transformation by Instrumented-Interpreter Specialization<sup>†</sup>

Søren Debois ([debois@itu.dk](mailto:debois@itu.dk))

*IT University of Copenhagen<sup>‡</sup>; Programming, Logic, and Semantics Group*

**Abstract.** We describe how to implement strength reduction, loop-invariant code motion and loop quasi-invariant code motion by specializing instrumented interpreters. To curb code duplication intrinsic to such specialization, we introduce a new program transformation, *rewinding*, which uses Moore-automata minimization to remove duplicated code.

**Keywords:** Partial evaluation, interpretive approach, imperative program transformation, code duplication, Moore automata.

**CCS Codes:** D.3.4, F.3.2.

## 1. Introduction

In this paper, we describe how to implement the imperative program transformations strength reduction, loop-invariant code motion, and loop quasi-invariance hoisting (Song et al., 2000) by specializing instrumented interpreters. Further, we introduce a novel program transformation, *rewinding*, which uses Moore-automata minimization to remove code duplication.

This contribution is significant for at least the following reasons. First, we provide simple alternative implementations for well-known, non-trivial imperative program transformations. Second, implementing program transformation by instrumented interpreter specialization, also known as the interpretive approach (Glück and Jørgensen, 1994), is not well-studied for imperative languages. We provide insight into the strengths and weaknesses of the interpretive approach in this setting. Finally, we provide both theoretical insight into the nature of code duplication as well as rewinding as a practical remedy.

Our specializer is standard except for the rewinding transformation, which is straightforward to implement as a variant of Moore automata minimization. The relative simplicity of our interpreters thus suggests that constructing and specializing instrumented interpreters might be a feasible way to construct optimizing compilers for domain-specific

---

<sup>†</sup> This paper is a revised and extended version of the conference paper (Debois, 2004).

<sup>‡</sup> Rued Langgaardsvej 7, 2300 Copenhagen S, Denmark



languages. In this way our results complement recent findings (Thibault et al., 2000) that compiling domain-specific languages by specialization can yield target programs performing comparably to hand-coded low-level programs.

How does the interpretive approach apply in an imperative setting? We instrument an interpreter to collect statically available information as it runs, thus establishing an *execution history*, which is subsequently used to justify transformations. For example, an interpreter may perform loop-invariant code motion on an assignment  $x := \langle \text{exp} \rangle$  by checking whether a particular execution of the assignment could change the value of  $x$ . If it would not, the interpreter simply skips the assignment. The interpreter uses the execution history to approximate whether executing the assignment would change  $x$ .

If a program point is reachable in more than one way, there may be more than one possible history for it. Thus, polyvariant specialization of history-collecting interpreters sometimes introduces code duplication, much the same way that polyvariant specialization of programs with dead static variable does (Gomard and Jones, 1991; Jones et al., 1993; Jones, 2004). This duplication is intrinsic to our methods; we cannot prevent it. Instead, we apply our rewinding transformation to the residual program, thereby removing duplication. Rewinding of a program proceeds by translating the program into a Moore automaton, minimizing that Moore automaton, and translating the minimized automaton back into a program. We shall see (Corollary 15, page 23) that rewinding preserves semantics.

We believe that ours is the first substantial application of the interpretive approach to an imperative language (although (Thibault et al., 2000) contain an application to a domain-specific imperative language). For functional languages, the interpretive approach has been used to do constant propagation and higher-order removal (Sperber and Thiemann, 1996), super-compilation (Glück and Jørgensen, 1994; Glück and Jørgensen, 1994) and deforestation (Glück and Jørgensen, 1994).

We expect the reader to have a basic proficiency in partial evaluation, corresponding to the level of (Jones et al., 1993). A passing familiarity with the interpretive approach (Glück and Jørgensen, 1994) and automata theory (Hopcroft and Ullman, 1979) would be helpful, but is not required.

This paper is organized as follows. In Section 2, we introduce the Flowchart language, a Flowchart interpreter, and a Flowchart specializer; in Section 3, we describe how to implement loop-invariant code motion and loop quasi-invariant hoisting; in Section 4, we describe how to implement strength reduction; in Section 5, we investigate code duplication; in Section 6, we define rewinding and prove it semantic

preserving; and in Section 7, we discuss our methods and give directions for future work.

## 2. Preliminaries

We use a variant of the Flowchart language of (Gomard and Jones, 1991; Jones et al., 1993). Syntax for the language is given in Figure 1;  $Id$  is the set of identifiers (variable names). A program is a list of input variables and a sequence of basic blocks, each of which is a series of assignments followed by a jump (`goto`), a conditional jump (`if`) or program termination (`return`). Notice that the flowchart language does not encompass pointer variables.

$$\begin{aligned}
 \textit{Program} &::= \textit{Input Block}^+ \\
 \textit{Input} &::= \textit{read Id}^* \\
 \textit{Block} &::= \textit{Label Asgns Jump} \\
 \textit{Jump} &::= \textit{goto Label} \\
 &\quad | \textit{if Param then Label else Label} \\
 &\quad | \textit{return Param} \\
 \textit{Asgns} &::= \textit{Asgn}^* \\
 \textit{Asgn} &::= \textit{Id := Expr} \\
 \textit{Expr} &::= \textit{Op Param}^+ \\
 &\quad | \textit{Param} \\
 \textit{Param} &::= \textit{Id} \\
 &\quad | \textit{' SExp} \\
 \textit{SExp} &::= \textit{atom} \\
 &\quad | (\textit{SExp}^*) \\
 \textit{Op} &::= \textit{hd} \mid \textit{tl} \mid \textit{cons} \mid + \mid - \mid = \mid < \mid \dots
 \end{aligned}$$

Figure 1. The Flowchart Language

We assume throughout that there are no jumps to undefined labels, that the label of each basic block is unique, and that each operator application has the correct number of parameters. We allow as syntactic sugar `if`, `while`, and `case`; compound expressions; infix `cons` operator `::`; short-circuiting boolean expressions; macros; labels inside basic blocks; and anonymous entry blocks. See Figure 4 for a simple example program.

Values for the Flowchart language are the S-expressions of LISP; that is, values are either atoms or pairs of values, atoms being either strings or numbers. The semantics of the Flowchart is as expected with execution commencing at the first basic block, the *entry* block.

*Definition 1. (Semantics of Flowchart)* Let  $P$  be a Flowchart program, and let  $l$  be a label of  $P$ . We write  $\llbracket l \rrbracket$  to mean the basic block of  $P$  with label  $l$ . Let  $V$  be some non-empty set of values with  $\mathbf{False} \in V$ . An *environment* is a finite map  $\sigma : Id \rightarrow V$ ; we write  $\mathcal{IE}$  for the set of all environments. We assume total functions  $\llbracket \cdot \rrbracket_A : \mathit{Asgns} \rightarrow \mathcal{IE} \rightarrow \mathcal{IE}$ , for executing assignments; and  $\llbracket \cdot \rrbracket_P : \mathit{Param} \rightarrow \mathcal{IE} \rightarrow V$ , for evaluating parameters. (The simple interpreter of Figure 3 implements  $\llbracket \cdot \rrbracket_A$  and  $\llbracket \cdot \rrbracket_P$  in the natural way.) Define a function  $\llbracket \cdot \rrbracket_J : \mathit{Jump} \rightarrow \mathcal{IE} \rightarrow \mathit{Label} \times \mathcal{IE}$  by

$$\begin{aligned} \llbracket \mathit{goto } l \rrbracket_J \sigma &= (l, \sigma) \\ \llbracket \mathit{if } p \mathit{ then } l_1 \mathit{ else } l_2 \rrbracket_J \sigma &= \begin{cases} (l_2, \sigma) & \text{if } \llbracket p \rrbracket_P \sigma = \mathbf{False} \\ (l_1, \sigma) & \text{otherwise} \end{cases} \\ \llbracket \mathit{return } p \rrbracket_J \sigma &= (\bullet, \sigma[Y \leftarrow \llbracket p \rrbracket_P \sigma]), \end{aligned}$$

where  $\bullet$  is different from all labels and  $Y$  is different from all variable names. Define a total function  $\llbracket \cdot \rrbracket_B : \mathit{Block} \rightarrow \mathcal{IE} \rightarrow \mathit{Label} \times \mathcal{IE}$  by  $\llbracket l \mathit{ Asgns } \mathit{ Jump} \rrbracket_B \sigma = \llbracket \mathit{Jump} \rrbracket_J (\llbracket \mathit{Asgns} \rrbracket_A \sigma)$ . A *run* of length  $n$  of  $P$  on environment  $\sigma_1$  is a sequence  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  such that  $l_i \neq \bullet$  for  $1 \leq i < n$ ,  $(l_i, \sigma_i) = \llbracket \llbracket l_{i-1} \rrbracket_B \sigma_{i-1} \rrbracket_B \sigma_{i-1}$  for  $1 < i \leq n$ , and  $l_1$  is the label of the entry block of  $P$ . We say that  $P$  *terminates on*  $\sigma_1$  *with value*  $v$  if there exists a run  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  such that  $l_n = \bullet$  and  $\sigma_n(Y) = v$ . We define a semantic function  $\llbracket P \rrbracket : \mathcal{IE} \rightarrow V_\perp$  by

$$\llbracket P \rrbracket \sigma = \begin{cases} v & \text{if } P \text{ terminates on } \sigma \text{ with value } v \\ \perp & \text{otherwise.} \end{cases}$$

Two programs  $P$  and  $P'$  are semantically equivalent iff  $\llbracket P \rrbracket = \llbracket P' \rrbracket$ .

We implement Flowchart interpreters in Flowchart itself extended with an indexed store construct, which is handy for implementing environments in Flowchart self-interpreters. The extended language, Flowchart<sup>+</sup>, has the two extra constructs shown in Figure 2: A new

$$\begin{array}{l} \vdots \\ \mathit{Asgn} ::= \dots \\ \quad | \llbracket \mathit{Param} \rrbracket := \mathit{Param} \\ \vdots \\ \mathit{Op} ::= \dots | \mathit{load} \end{array}$$

Figure 2. The Flowchart<sup>+</sup> Language

assignment form allows the assignment of a value to a store location,

and a new operator allows the retrieval of a value from a store location. We use ‘ $[Param]$ ’ as syntactic sugar for ‘ $load\ Param$ ’.

A Flowchart-interpreter written in Flowchart<sup>+</sup> is given in Figure 3. Programs are represented in the obvious manner: a program is a list of blocks, a block is a 3-tuple, and so forth. Execution commences at line 9; the first 8 lines include convenient macros (relegated to Appendix A) and define a macro for evaluating parameters. The interpreter moves the actual parameters into the store (lines 11–15) and proceeds to the basic-block executing loop (lines 18–63). The basic block to be executed is fetched and split into constituent parts (lines 19–20), whereafter the assignments of the current block are executed (lines 21–47) by evaluating the parameters (lines 26–32), and applying the appropriate operator to the values of the parameters (lines 33–45). Finally, the jump of the basic block is executed (lines 48–63). (The label `next-assign` is not used before we instrument the interpreter in the next section.)

*The Specializer* To do our experiments, we have implemented a rudimentary polyvariant, transition-compressing, program-point specializer from Flowchart<sup>+</sup> to Flowchart, as described in (Jones et al., 1993). We restrict our attention to Flowchart<sup>+</sup>-programs in which every store access is to a statically known location; this restriction ensures that store locations can be replaced by variable accesses in the residual program. Naturally, all of our interpreters are in this class of Flowchart<sup>+</sup> programs.

Our specializer applies the rewinding transformation after specialization proper, to remove code duplication in residual programs. (We discuss rewinding in detail in Sections 5 and 6.) All the residual programs exhibited in this paper are actual output of our specializer.

*A note on terminology* To avoid the awkward phrase “ $P$  is the result of specializing interpreter `int` w.r.t. program  $P$ ” we shall say “ $P'$  transformed by `int` yields  $P$ ”. This slight abuse of language emphasizes the intuition that an instrumented interpreter implements a transformation.

### 3. Loop-Invariant Code Motion

Loop-invariant code motion (Muchnick, 1997) identifies computations in loops that yield the same value on every iteration and moves such computations out of the loop. Consider the program in Figure 4. Given a positive natural number  $n$ , this program computes  $(2n - 1)!$ . The assignment `t := 2 * n` in line 5 is recomputed on each iteration of

```

1  include "macros.i"
macro eval-param (p) :
    case hd p of
4      'quote: return tl p;
        'id:    return [tl p];
        default: return '(malformed param);
    end;
8  end
read pgm, data;
    formals := tl (hd pgm);
    while formals != '() do
12      [hd formals] := hd data;
        formals := tl formals;
        data := tl data;
    end;
16  bbs := hd (tl pgm);
    next := hd (hd bbs);
exec-bb:
    bb := lookup (next, bbs);
20  asgns := hd bb; jump := tl bb;
    while asgns != '() do
        asgn := hd asgns; asgns := tl asgns;
24  id := hd asgn; exp := tl asgn;
        op := hd exp; params := tl exp;
        param1 := '(); param2 := '();
        if params != '() do
28  param1 := eval-param (hd params);
            params := tl params;
        end;
        if params != '() do
            param2 := eval-param (hd params);
32  end;
        case op of
            'hd: [id] := hd param1;
            'tl: [id] := tl param1;
36  'not: [id] := ! param1;
            'base: [id] := param1;
            'eq: [id] := param1 = param2;
            'cons: [id] := cons (param1,param2);
40  'add: [id] := param1 + param2;
            'mult: [id] := param1 * param2;
            'less: [id] := param1 < param2;
44  default:
            return '(bad operator);
        end;
    next-asgn:
    end;
48  case hd jump of
        'goto:
            next := tl jump;
        'return:
52  return eval-param (tl jump);
        'if:
            param := hd (tl jump);
            labtrue := hd (tl (tl jump));
56  labfalse := tl (tl (tl jump));
            if eval-param (param) do
                next := labtrue;
            else
60  next := labfalse;
            end;
        end;
    goto exec-bb;

```

Figure 3. A simple interpreter for Flowchart.

```

1  read n;
    f := 1;
    i := 1;
4  loop:
    t := 2 * n;
    again := i < t;
    if again then body else done;
8  body:
    f := f * i;
    i := i + 1;
    goto loop;
12 done:
    return f;

```

Figure 4. Program fac.

the loop, even though the value of `n` never changes. We shall see how transforming `fac` by an instrumented interpreter may yield the program in Figure 7.

Executing an assignment `x := <exp>` cannot change the value of `x` if (a) the previous assignment to `x` were also `x := <exp>` and (b) the variables used in `<exp>` were not updated in the meantime. If these conditions are satisfied, we say that the assignment `x := <exp>` is *available*.

Maintaining a list of available assignments is easy in an interpreter. Every time an assignment `x := <exp>` has been executed, we remove all assignments that use or define `x` from the available assignments list. Then we add the assignment `x := <exp>` to the list, provided `<exp>` does not reference `x`. The available assignments are a subset of the assignments of the input program, so the available assignments list is of bounded static variation (Jones et al., 1993). Thus, recording available assignments does not introduce extra specialization-time non-termination in the interpreter.

A code fragment maintaining the available assignments list is given in Figure 5. The fragment is inserted immediately after the code for

```

1  tmp := '();
    while available != '() do
      asgn' := hd available;
4    if !(member (id, use (asgn'))) || def (asgn') = id do
      tmp := cons (asgn', tmp);
      end;
      available := tl available;
8  end;
    available := reverse (tmp);
    if !(member (id, use (asgn'))) do
      available := cons (asgn, available);
12 end;

```

Figure 5. Implementation of available assignments analysis.

execution of an assignment — that is, between line 45 and line 46 in Figure 3. Suppose an assignment `id := <exp>` has just been exe-

cuted. First, any previously available assignments that use or define `id` are removed from the list (lines 1–9). Then the current assignment is added to the list of available assignments, provided it is not a self-assignment (lines 10–12). The macros `reverse`, `member`, `use` and `def` compute list reversal, list membership, the list of variables referenced in an assignment and the variable defined by an assignment, respectively.

The code fragment in Figure 6 skips execution of superfluous assignments. It is inserted immediately before the code for assignment execution — that is, between line 24 and line 25 of Figure 3. The fragment searches the list of available assignments for the current assignment, which is skipped if found.

```

1      tmp := available;
      while tmp != '()' do
          asgn' := hd tmp;
4          if asgn = asgn' do goto next-asgn; end;
          tmp := tl tmp;
      end;

```

Figure 6. Implementation of superfluous assignment skipping.

Augmenting the simple interpreter of Figure 3 with these fragments, we obtain the *hoisting* interpreter. Transforming the `fac` program of Figure 4 by the hoisting interpreter yields the program in Figure 7. The non-contiguous label names L0, L3 and L4 are an artifact of our

```

1  read data;
   L0:      n := hd data;
4      f := 1;
        i := 1;
        t := 2 * n;
        again := i < t;
8      if again then L3 else L4;
   L3:      f := f * i;
        i := i + 1;
12     again := i < t;
        if again then L3 else L4;
   L4:      return f;

```

Figure 7. Transformation of `fac` (Figure 4) by the hoisting interpreter.

specializer applying rewinding to remove code duplication; refer to Section 5 for details.

The test part of the loop (lines 5–7 of the original program in Figure 4) has been unrolled once in the residual program in Figure 7: in the first iteration (lines 6–8), `t := 2 * n` is computed; in subsequent iterations (lines 12–13), it is skipped. This unrolling occurs because the value of the variable `available` is static: Each basic block of the input program occurs in the output program once for each possible

value of `available`. For the `fac` program, the value of `available` at the first iteration of `loop` is

```
[i := 1, f := 1],
```

whereas on the second iteration onwards it is

```
[t := 2 * n].
```

(The statement `again := i < t` is not available because of subsequent assignment to `i`; similarly, the statements `f := f * i` and `i := i + 1` are not available because they are self-assignments.) Thus, the `loop` block occurs twice in the output: in lines 5–8 and lines 12–13.

However, the assignment `t := 2 * n` appears only in the first occurrence, not in the second. On the second iteration of `loop`, the interpreter skips the execution of `t := 2 * n` precisely because it is in `available` (see Figure 6); hence, that statement does not occur in the output program.

The transformation achieved by the hoisting interpreter is slightly different from what is usually understood by loop-invariant code motion: Rather than moving the invariant assignment `t := 2 * n` outside of the loop, we have instead unrolled the loop, in the process removing the redundant computation from the second iteration onwards. Unrolling the loop reduces loop-invariant code motion to a trivial case of common-subexpression elimination.

This implementation also does *loop quasi-invariant hoisting* (Song et al., 2000). This is a loop optimization in which one recognizes that after some finite number of iterations, an assignment in the loop becomes invariant. Figure 8 contains an example. The basic blocks `loop`

```

1  read n;
   i := 0;
   sum := 0;
4   x1 := 0;
   x2 := 0;
   loop:
8   t := i < n;
   if t then body else done;
   body:
   x3 := x2 + 1;
   x2 := x1 + 1;
12  x1 := 2 * n;
   sum := sum + x3;
   i := i + 1;
   goto loop;
16  done:
   return sum;
```

Figure 8. Opportunity for loop quasi-invariance code motion.

and `body` forms a loop, in which `x1` is constant from the second iteration, `x2` is constant from the third iteration, and `x3` is constant from the

iteration	available
1	[x2 := 0, x1 := 0, sum := 0, i := 0]
2	[x1 := 2 * n]
3	[x2 := x1 + 1, x1 := 2 * n]
4	[x3 := x2 + 1, x2 := x1 + 1, x1 := 2 * n]

Figure 9. The value of `available` at the block loop.

fourth iteration. We would expect a clever optimizing compiler to unroll the loop precisely four times, successively removing the loop-invariant computations of `x1`, `x2` and `x3`.

Our implementation does loop quasi-invariance hoisting. On the second iteration of the loop, the hoisting interpreter skips `x1 := 2 * n` because it is available. Thus `x1` is not modified, so `x2 := x1 + 1` is available on the third iteration. Similarly, `x3` is available on the fourth iteration, on which `available` stabilizes. (The values of `available` for `loop` are listed in Figure 3.) Thus, the block `loop` occurs four times in the output: once for each value of `available`.

Transforming the program of Figure 8 by the hoisting interpreter yields the program in Figure 10. The loop has been unrolled four times with the assignments to `x1`, `x2` and `x3` successively eliminated.

Although the program of Figure 8 is admittedly contrived, it is suggested in (Song et al., 2000) that opportunities for loop quasi-invariant hoisting are not uncommon in automatically generated programs.

#### 4. Strength Reduction

Consider the program `div` of Figure 11. This program computes the function

$$f(n, e) = 3 \left\lceil \frac{n}{3e} \right\rceil, \quad n, e \in \mathcal{N},$$

where  $\lceil x \rceil$  is the least integer greater than or equal to  $x$ . The variable `j` will assume the values  $0, 3e, 6e, 9e, \dots$ , so we may replace line 9 with `j := j + (3 * e)`. The expression `3 * e` is loop-invariant, so if we compute it outside of the loop and store the result in some fresh variable, say, `v0 := 3 * e`, we can change line 9 to `j := j + v0`. If addition is faster than multiplication, this transformation may be an optimization. Automatically converting multiplications to additions is called *strength reduction* (Muchnick, 1997).

In the above example, we replace `i * e` with `j + (3 * e)`. Towards instrumenting an interpreter to do strength reduction, we consider the

```

1  read data;
   L0:
4      n := hd data;
       i := 0;
       sum := 0;
       x1 := 0;
       x2 := 0;
8      t := i < n;
       if t then L1 else L4;
   L1:
12     x3 := x2 + 1;
       x2 := x1 + 1;
       x1 := 2 * n;
       sum := sum + x3;
       i := i + 1;
       t := i < n;
16     if t then L3 else L4;
   L3:
20     x3 := x2 + 1;
       x2 := x1 + 1;
       sum := sum + x3;
       i := i + 1;
       t := i < n;
24     if t then L5 else L4;
   L4:
28     return sum;
   L5:
       x3 := x2 + 1;
       sum := sum + x3;
       i := i + 1;
       t := i < n;
32     if t then L7 else L4;
   L7:
36     sum := sum + x3;
       i := i + 1;
       t := i < n;
       if t then L7 else L4;

```

Figure 10. Transformation of the program of Figure 8 by the hoisting interpreter.

```

1  read n, e;
   i := 0;
   j := 0;
4  loop:
   t := j < n;
   if t then body else done;
   body:
8     i := i + 3;
       j := i * e;
       goto loop;
12 done:
   return i;

```

Figure 11. Program div.

relationship of these variables when we are about to evaluate  $i * e$ . Using  $i$ ,  $e$  and  $j$  to denote the values of  $i$ ,  $e$  and  $j$ , respectively, and using  $i'$  to denote the previous value of  $i$ , we find

$$\begin{aligned} j &= i'e \\ i &= i' + 3 \end{aligned} \tag{1}$$

Thus

$$ie = (i' + 3)e = i'e + 3e = j + 3e. \tag{2}$$

Because  $i * e$  evaluates to  $ie$ , we may substitute  $j + (3 * e)$  for it, provided the *values* of  $i$ ,  $j$  and  $e$  are related by some value  $i'$  as specified in (1). Obviously, we can generalize the “3” to an arbitrary constant  $c$ . In this case

$$\begin{aligned} j &= i'e \\ i &= i' + c \end{aligned} \tag{3}$$

and

$$ie = (i' + c)e = i'e + ce = j + ce.$$

Thus, for any constant  $c$ , we may replace  $i * e$  with  $j + (c * e)$ , provided the values  $i$ ,  $e$  and  $j$  are related by some  $i'$  as specified by (3). This condition is quite general; for example, it says neither that  $i$  and  $j$  should occur within a loop, nor that the value  $i'$  should be the previous value of  $i$ .

We restrict our attention to a simple case sufficient to do strength reduction on the `div` program. Consider a sequence of assignments

$$j := i * e, \dots, i := i + c, \dots, j := i * e. \tag{4}$$

Any such sequence produces values satisfying (3), provided that  $e$  is constant throughout it and that it contains no other definitions of  $i$  and  $j$ . By recording a history of recent assignments, an interpreter can check whether the assignments executed so far constitute such a sequence; we call this history the *reaching definitions*. The reaching definitions differ from the available assignments of the hoisting interpreter in that they may contain assignments that are no longer available.

The interpreter variable that stores the reaching definitions must be of bounded static variation. We ensure this boundedness by recording at most one assignment for each variable of the input program, replacing older assignments with newer ones as we go.

The actual strength reduction is performed by the interpreter replacing the statement  $j := i * e$  with the statements  $v0 := c * e$  and  $j := j + v0$ , where  $v0$  is some fresh variable name. Loop-invariant hoisting can then move the former statement out of the loop.

When  $j := i * e$  has been strength reduced to  $j := j + e$ , we must be careful to put the *original* assignment  $j := i * e$  in the reaching definitions. If we insert the strength-reduced alternative assignments, then on the next iteration of the loop, the interpreter will find the reaching definitions sequence

$$j := j + e, \dots, i := i + c, \dots, j := i * e.$$

This sequence does not have the form (4), so there will be no strength reduction. In the implementations below, the `revision` variable records the necessity of revising the reaching definitions.

Strength reduction is implemented in Figure 12 (collecting reaching definitions) and 13 (recognizing sequences of the form (4) and actually performing strength reduction). The former is to be inserted along with the available assignments computation (Figure 5) between line 45 and line 46 of the simple interpreter (Figure 3), the latter is to be inserted along with the hoisting implementation (Figure 6) between line 25 and line 26.

```

1      if revision != '()' do
        asgn' := hd revision;
        revision := tl revision;
4      else
        asgn' := asgn;
        end;
        left := reaching;
8      right := '()';
        while left != '()' do
          asgn'' := hd left;
          if def (asgn'') = def (asgn') then
12         cutoff;
            right := cons (asgn'', right);
            left := tl left;
        end;
16     cutoff:
        reaching := asgn' :: reverse (right);

```

Figure 12. Implementation of reaching definitions analysis.

The implementation of strength reduction in Figure 13 derives its complexity mostly from the unsuitability of Flowchart for list processing. The implementation simply scans the reaching definitions, starting at the most recent one, checking if they are on the form (4).

Note the construction of a name for the necessary temporary variable in line 36. Exploiting that variable names, as represented in the interpreter, are not restricted to atoms (i.e., the value of a `cons`-expression is a legal variable name), the interpreter produces a name for the temporary by concatenating the label of the current basic block with the assignments remaining in the current basic block. Together, these two uniquely identify an assignment in the input program. (When outputting the residual program, our partial evaluator converts non-atomic variable names to strings of the form  $vn$  not otherwise present in the residual program.)

Altogether, we have constructed the *strength-reducing* interpreter. Transforming the `div` program by the strength-reducing interpreter yields the program of Figure 14. Note the unrolling of the main loop: the first iteration establishes the preconditions, the second iteration computes the loop-invariant multiplication, and subsequent iterations are strength reduced.

```

1   if hd exp = 'mult &&
    hd (param10f (asgn)) = 'id &&
    hd (param20f (asgn)) = 'id
4   do
    j := def (asgn);
    i := tl (param10f (asgn));
    e := tl (param20f (asgn));
8   rs := reaching;
    while rs != '() do
    asgn' := hd rs; rs := tl rs;
    if i = def (asgn') do
12   if 'add = hd (tl asgn') &&
    ('id :: i) = param10f (asgn') &&
    'quote = hd (param20f (asgn'))
    do
16   c := tl (param20f (asgn'));
    goto found-incr;
    else
20   goto abort-strength-reduction;
    end;
    end;
    if j = def (asgn') || e = def (asgn') then
24   abort-strength-reduction;
    end;
    goto abort-strength-reduction;
found-incr:
28   while rs != '() do
    asgn' := hd rs; rs := tl rs;
    if asgn = asgn'
    then do-strength-reduction;
    if i = def (asgn') || e = def (asgn')
32   then abort-strength-reduction;
    end;
    goto abort-strength-reduction;
do-strength-reduction:
36   tmp := next :: asgns;
    mult := mkBinOpAsgn (tmp, 'mult,
    'quote :: c,
    'id :: e);
40   asgn' := mkBinOpAsgn (j, 'add,
    'id :: j,
    'id :: tmp);
    asgns := mult :: asgn' :: asgns;
44   revision := mult :: asgn :: '();
    goto next-asgn;
abort-strength-reduction:
    end;

```

Figure 13. Implementation of strength reduction.

## 5. Code Duplication

Consider the program `count` of Figure 15. Given an integer  $n$  and an integer list  $xs$ , this program counts the number of elements of  $xs$  greater than  $n$ .

If we transform the `count` program by the hoisting interpreter, but omit post-specialization rewinding, we get the program in Figures 16 and 17; the latter figure represents the program graphically, solid edges corresponding to `then`-branches and dashed edges to `else`-branches.

```

1  read data;
   L0:
4      n := hd data;
       data := tl data;
       e := hd data;
       i := 0;
       j := 0;
8      t := j < n;
       if t then L1 else L2;
   L1:
12     i := i + 3;
       j := i * e;
       t := j < n;
       if t then L3 else L2;
   L2:
16     return i;
   L3:
20     i := i + 3;
       v0 := 3 * e;
       j := j + v0;
       t := j < n;
       if t then L9 else L2;
24    L9:
       i := i + 3;
       j := j + v0;
       t := j < n;
       if t then L9 else L2;

```

Figure 14. Transformation of div (Figure 11) by the strength-reducing interpreter.

```

1  read n, xs;
   start:
4      i := 0;
       goto loop;
   loop:
8      t := xs = '();
       if t then done else body;
   body:
12     x := hd xs;
       xs := tl xs;
       t := n < x;
       if t then inc else loop;
   inc:
16     i := i + 1;
       goto loop;
   done:
       return i;

```

Figure 15. Program count.

The transformed program clearly contains redundant blocks. Comparing it to the original, we see that the block L0 corresponds to the concatenation of the `start` and `loop` blocks, modulo labels and argument-reading artifacts of the interpreter. This concatenation is a result of transition compression. Moving on, L1 corresponds to `done`, L2 to `body`, L3 to the concatenation of `inc` and `loop` (again concatenated by transition compression) and L4 to `loop`. Altogether, the blocks L0 through L4 correspond to the entire input program, so we would prefer that there were no more blocks in the transformed program. Alas, there are.

```

1  read data;
   L0:      n := hd data;
4         data := tl data;
          xs := hd data;
          i := 0;
          t := xs = '();
8         if t then L1 else L2;
   L1:      return i;
   L2:      x := hd xs;
12         xs := tl xs;
          t := n < x;
          if t then L3 else L4;
   L3:      i := i + 1;
          t := xs = '();
          if t then L5 else L6;
20        L4:      t := xs = '();
          if t then L1 else L2;
   L5:      return i;
   L6:      x := hd xs;
24         xs := tl xs;
          t := n < x;
          if t then L7 else L8;
   L7:      i := i + 1;
32         t := xs = '();
          if t then L5 else L6;
   L8:      t := xs = '();
36         if t then L5 else L6;

```

Figure 16. Transformation of `count` (Figure 15) by the hoisting interpreter, omitting post-specialization rewinding.

The blocks L3 and L4 both end with code corresponding to `loop`, so we would expect L3 and L4 to branch to the same continuations. However, they do not. Whereas the L4 block returns to either L1 or L2, the L3 block branches to either L5 and L6.

Both L1 and L5 correspond to `done`, and both L2 and L6 correspond to `body`. Similarly, both L7 and L3 correspond to the concatenation of `inc` and `loop`, and both L8 and L4 correspond to `loop` block.

Altogether, the transformation of `count` by the hoisting interpreter has caused the main loop formed by `loop`, `body` and `inc` to be duplicated into the blocks L1 through L4 and the blocks L5 through L8.

Why is this? The available assignments at the `loop` block depend on the path taken to reach `loop`. If we arrive via `start`, then the assignment `i := 0` is available. If we arrive via `inc`, then no assignment to `i` is available, because of the self-assignment `i := i + 1` in `inc`. Finally, if we arrive via `body`, then the available assignments for `i` depends on the path taken to reach the `body` block, because the `body` block does not itself change the availability of `i`.

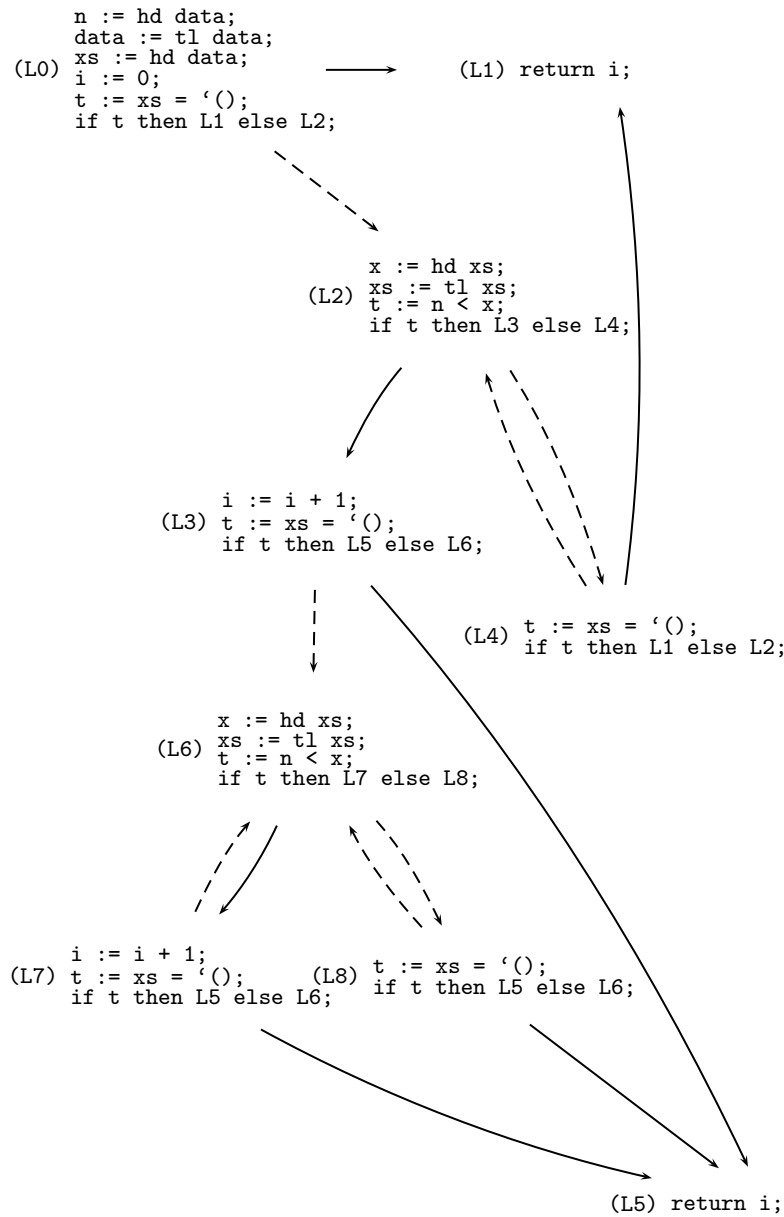


Figure 17. Flowgraph representation of the program of Figure 16.

Although the difference in available assignments does not affect the hoisting interpreter's subsequent actions, specialization is still performed once for each possible history — hence, code duplication. In the blocks L1 through L4, the assignment  $i := 0$  is available, whereas in L5 through L8, no assignment for  $i$  is available.

The histories for  $i$  follow a pattern common to most loops: a variable is initialized to some value (one history), and subsequently updated for each iteration of a loop (a second history). The resulting code duplication is compounded by consecutive loops, possibly leading to a residual program exponentially larger than the original program.

This duplication has the flavor of the dead static variable problem (Gomard and Jones, 1991; Jones, 2004; Jones et al., 1993). Whereas the dead static variable problem is easily solved by reclassifying dead static variables as dynamic, there is no simple way to avoid recording useless available assignments without defeating the purpose of easily implementing transformations. Thus, we settle for removing redundancy after specialization.

Looking at the duplicated blocks L1 and L5, we see that they are equivalent in the sense that they execute the same assignments, perform the same test, and branch to similarly equivalent blocks. This notion of equivalence resembles equivalence of states in Moore automata (i.e., deterministic finite automata with labelled states). In turn, this resemblance hints at a method for removing code duplication in a program  $P$ : translate  $P$  to a Moore automaton, minimize that automaton, and translate the minimized automaton back to a program  $P'$ . We call this procedure *rewinding*. Without rewinding, the output programs exhibited in Sections 3 and 4 would contain various degrees of duplication.

In the next section, we define these translations; prove that they commute with semantic equivalence and automata equivalence, respectively; and define rewinding formally. But first, let us see rewinding in action: Applying the rewinding transformation to the residual program in Figure 16 and 17 yields the program in Figure 18. Note how the blocks L1 – L4 have been removed.

## 6. Rewinding & Moore Automata

In this section we translate programs to Moore automata and vice versa. The encodings are mutually inverse and ensure that equivalence of Moore automata entail semantic equivalence of their corresponding programs.

```

1  read data;
   L0:      n := hd data;
4         data := tl data;
          xs := hd data;
          i := 0;
          t := xs = '();
8         if t then L5 else L6;
   L5:      return i;
12        L6:      x := hd xs;
          xs := tl xs;
          t := n < x;
          if t then L7 else L8;
16        L7:      i := i + 1;
          t := xs = '();
          if t then L5 else L6;
20        L8:      t := xs = '();
          if t then L5 else L6;

```

Figure 18. Transformation of `count` (Figure 15) by the hoisting interpreter, including post-specialization rewinding.

We begin by recalling the definition of a Moore automaton (Hopcroft and Ullman, 1979). Intuitively, a Moore automaton is a deterministic finite automaton where each state has a label, which is output upon visiting that state.

*Definition 2. (Moore automaton)* A Moore automaton is a six-tuple  $(S, \Sigma, \Delta, \delta, \lambda, s_0)$  of finite states  $S$ , input alphabet  $\Sigma$ , output alphabet  $\Delta$ , transition function  $\delta : S \times \Sigma \rightarrow S$ , labeling function  $\lambda : S \rightarrow \Delta$  and initial state  $s_0$ . We extend transition functions  $\delta$  to sequences of input symbols by taking  $\delta(\epsilon) = s_0$  and  $\delta(a_1 \dots a_n) = \delta(\delta(a_1 \dots a_{n-1}), a_n)$ .

The meaning of a Moore automaton  $m$  is the function  $\llbracket m \rrbracket : \Sigma^* \rightarrow \Delta^+$  given by  $\llbracket m \rrbracket(\epsilon) = \lambda(s_0)$  and  $\llbracket m \rrbracket(a_1 \dots a_n) = \llbracket m \rrbracket(a_1 \dots a_{n-1}) \cdot \lambda(\delta(a_1 \dots a_n))$ . Moore automata  $m, m'$  are equivalent iff  $\llbracket m \rrbracket = \llbracket m' \rrbracket$ .

The encodings  $\phi$  (programs to Moore automata) and  $\phi^{-1}$  (Moore automata to programs) will be given in Definitions 9 and 10, respectively. First, we define rewinding and state its correctness. We remove code duplication in a program  $P$  by translating  $P$  to a Moore automaton, minimizing that automaton, and translating the minimized automaton back to a program  $P'$ ; this procedure is rewinding<sup>1</sup>.

<sup>1</sup> This formulation of rewinding is equivalent to the one given in (Debois, 2004): Simply observe that for deterministic automata, language equivalence coincides with bisimulation equivalence. Indeed, implementing rewinding as defined in the present paper, but suppressing the translation to automata (working instead directly on the basic blocks) would lead to the same implementation as the one for rewinding as defined in (Debois, 2004). Hence, we have simply reused the latter implementation

*Definition 3. (Rewinding)* Let  $\min$  be the function that minimizes Moore automata. Given a program  $P$ , the *rewinding of  $P$*  is the program  $\phi^{-1}(\min(\phi(P)))$ .

To prove that rewinding preserves semantics, we will need the following theorem.

**THEOREM 4.** *If  $m$  and  $m'$  are equivalent Moore automata, then their translations  $\phi^{-1}(m)$  and  $\phi^{-1}(m')$  are semantically equivalent programs.*

Because minimization of a Moore automaton yields an equivalent automaton (Denning et al., 1978), Theorem 4 implies that rewinding is a semantics-preserving transformation.

**COROLLARY 5** (Correctness of rewinding). *Let  $P$  be any program  $P$ . Then  $P$  and the rewinding  $\phi^{-1}(\min(\phi(P)))$  of  $P$  are semantically equivalent.*

*Proof.* By definition, the Moore automata  $\min(\phi(P))$  and  $\phi(P)$  are equivalent. Thus, by Theorem 4, the programs  $\phi^{-1}(\min(\phi(P)))$  and  $P$  are semantically equivalent.

The remainder of this section serves to define the translations  $\phi$  and  $\phi^{-1}$  and to prove Theorem 4. We will need some notation before we can define the translations  $\phi$  and  $\phi^{-1}$ . First, it will prove convenient to have names for the labels in the branch targets of a basic block.

*Definition 6. (Successors)* Let  $b$  be a basic block and let  $\text{Jump}$  be the jump of  $b$ . We define the 0-, 1- and (-)-successor of  $b$  by

$$\begin{aligned} \text{succ}(b, -) &= l \text{ iff } \text{Jump} = \text{goto } l. \\ \text{succ}(b, 0) &= l \text{ iff } \text{Jump} = \text{if } p \text{ then } x \text{ else } l \\ \text{succ}(b, 1) &= l \text{ iff } \text{Jump} = \text{if } p \text{ then } l \text{ else } x \end{aligned}$$

Code duplication results in basic blocks that differs only in their labels. To capture such duplication, we abstract away from actual labels in our automata representation of programs, using automata transitions rather than textual labels to represent control flow. We make precise the notion of “abstracting away from actual labels” in the following notion of “anonymization” and “anonymous basic blocks”.

---

for the present paper. The reason we have chosen a different formulation at all is that the automata-based definition catches the intuition behind the transformation explicitly: it states that we abstract away from labels.

*Definition 7. (Anonymous Basic Block)* Let  $b$  be a basic block, let  $\text{Asgn}$  be the assignments of  $b$  and let  $\text{Jump}$  the jump of  $b$ . We define the *anonymization*  $\mathcal{A}(b)$  of  $b$  by

$$\mathcal{A}(b) = \begin{cases} \text{Asgn goto} & \text{if } \text{Jump} = \text{goto } l' \\ \text{Asgn return } p & \text{if } \text{Jump} = \text{return } p \\ \text{Asgn if } p & \text{if } \text{Jump} = \text{if } p \text{ then } l' \text{ else } l''. \end{cases}$$

We call the range of  $\mathcal{A}$  the set of *anonymous basic blocks*. We lift  $\mathcal{A}$  to sequences by taking  $\mathcal{A}(b_1 \dots b_n) = \mathcal{A}(b_1) \dots \mathcal{A}(b_n)$  and call such sequences *anonymous execution sequences*. We lift  $\mathcal{A}$  to labels by taking  $\mathcal{A}(l) = \mathcal{A}([l])$ .

To go from automata to programs, we will need to a way to reconstruct a basic block from an anonymous basic block given the missing labels.

*Definition 8. (Reconstruction)* Given a label  $l$ , an anonymous basic block  $a$  with assignments  $\text{Asgn}$ , and a successor function  $\delta$ , define the *reconstruction*  $\mathcal{R}(l, a, \delta)$  by

$$\mathcal{R}(l, a, \delta) = \begin{cases} l \text{ Asgn goto } \delta(l, -) & \text{if } a = \text{Asgn goto} \\ l \text{ Asgn return } p & \text{if } a = \text{Asgn return } p \\ l \text{ Asgn if } p \text{ then} \\ \quad \delta(l, 0) \text{ else } \delta(l, 1) & \text{if } a = \text{Asgn if } p. \end{cases}$$

Note that  $\mathcal{R}$  is defined only insofar  $\delta$  is defined.

We now have sufficient notation to define translations between programs and Moore automata. We consider only Moore automata whose states  $S$  are a subset of the set of all Flowchart labels, whose input alphabet is  $\Sigma = \{0, 1, -\}$ , and whose output alphabet  $\Delta$  is the set of all anonymous basic blocks.

*Definition 9. (Programs to Moore automata,  $\phi$ )* Let  $P$  be a program. The Moore automaton  $\phi(P)$  has the labels of  $P$  as states, the function  $\text{succ}$  as the transition function, the anonymization function  $\mathcal{A}$  as labeling function and the label  $l_e$  of the entry state as initial state.

*Definition 10. (Moore automata to programs,  $\phi^{-1}$ )* Suppose  $m$  is a Moore automaton with  $m = (S, \Sigma, \Delta, \delta, \lambda, s_0)$ . Define  $\delta'$  by  $\delta'(l, t) = \delta(l, t)$  whenever the latter is defined and  $\delta'(l, t) = s_0$  otherwise. For each state  $l \in S$ , the program  $\phi^{-1}(m)$  has the basic block  $\mathcal{R}(l, \lambda(l), \delta')$ ; the entry block of  $\phi^{-1}(m)$  is the one labeled by  $s_0$ .

Note that both  $\phi$  and  $\phi^{-1}$  are computable.

LEMMA 11. *The translations  $\phi$  and  $\phi^{-1}$  are mutually inverse, that is, for all programs  $P$  and Moore automata  $m$ , both  $\phi^{-1}(\phi(P)) = P$  and  $\phi(\phi^{-1}(m)) = m$ .*

*Proof.* Immediate from Definitions 9 and 10.

To prove Theorem 4, we will need notation for the outcome of a conditional branch.

*Definition 12. (Jump)* Let  $b$  be a basic block and  $l$  a label. We define the *jump from  $b$  to  $l$* , written  $\mathcal{J}(b, l)$  by

$$\mathcal{J}(b, l) = \begin{cases} 0 & \text{if succ}(b, 0) = l \\ 1 & \text{if succ}(b, 1) = l \\ - & \text{if succ}(b, -) = l. \end{cases}$$

We lift  $\mathcal{J}$  to sequences of labels by taking  $\mathcal{J}(l) = \epsilon$  and  $\mathcal{J}(l_1 \dots l_n) = \mathcal{J}(l_1 \dots l_{n-1})\mathcal{J}(\lfloor l_{n-1} \rfloor, l_n)$ .

To ensure well-definedness of  $\mathcal{J}$ , we will assume that no program contains a jump *if  $p$  then  $l$  else  $l$* . This is not a restriction; such jumps are equivalent to *goto  $l$* , because we stipulated in Definition 1 that the function  $\llbracket \cdot \rrbracket_P$ , which evaluates the  $p$ , is total.

We now have enough notation. Towards Theorem 4, the following Lemma states that if two basic blocks execute the same assignment and test, then they modify their environment in the same way and take the same branches.

LEMMA 13. *Let  $b$  and  $b'$  be basic blocks such that  $\mathcal{A}(b) = \mathcal{A}(b')$  and  $\llbracket b \rrbracket_B \sigma = (l, \sigma')$ . There exists a label  $l'$  such that  $\llbracket b' \rrbracket_B \sigma = (l', \sigma')$  and  $\mathcal{J}(b, l) = \mathcal{J}(b', l')$ .*

*Proof.* Immediate from Definitions 1, 7 and 12.

We now prove that the transitions of a Moore automaton corresponding to some program adequately mimics that program's control flow, i.e., whenever the program branches on, say, true from  $l_i$  to  $l_j$ , the Moore automaton has a 1-transition from  $l_i$  to  $l_j$  and so forth.

LEMMA 14. *Let  $P$  be a program and let  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  be a run of  $P$ . Then  $\delta_{\phi(P)}(\mathcal{J}(l_1 \dots l_n)) = l_n$ .*

*Proof.* By induction on  $n$ . For the base case  $n = 1$ , observe that by definition of run,  $l_1$  must be the entry block of  $P$ , hence  $\delta_{\phi(P)}(\mathcal{J}(l_1)) = \delta_{\phi(P)}(\epsilon) = l_1$ . For the inductive step, assume that  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  is a run. By induction  $\delta_{\phi(P)}(\mathcal{J}(l_1 \dots l_{n-1})) = l_{n-1}$ . Assume  $\mathcal{J}(l_1 \dots l_n) =$

$t_1 \dots t_{n-1}$ . We compute

$$\begin{aligned}
\delta_{\phi(P)}(\mathcal{J}(l_1 \dots l_n)) &= \delta_{\phi(P)}(t_1 \dots t_{n-1}) \\
&= \delta_{\phi(P)}(\delta_{\phi(P)}(t_1 \dots t_{n-2}), t_{n-1}) \\
&= \delta_{\phi(P)}(\delta_{\phi(P)}(\mathcal{J}(l_1 \dots l_{n-1})), t_{n-1}) \\
&= \delta_{\phi(P)}(l_{n-1}, t_{n-1}) \\
&= l_n.
\end{aligned}$$

The last equality follows from the definition of  $\phi(P)$ .

It follows that each abstract execution sequence corresponds to the output of an automaton.

**COROLLARY 15.** *Suppose that  $P$  is a program and that there is a run  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  of  $P$ . Then  $\mathcal{A}(l_1 \dots l_n) = \llbracket \phi(P) \rrbracket (\mathcal{J}(l_1 \dots l_n))$ .*

*Proof.* It is sufficient to note that for  $1 \leq i \leq n$ ,

$$\mathcal{A}(l_i) = \mathcal{A}(\lfloor l_i \rfloor) = \lambda_{\phi(P)}(l_i) = \lambda_{\phi(P)}(\delta_{\phi(P)}(\mathcal{J}(l_1 \dots l_i))).$$

We can now prove that equivalent Moore automata yield equivalent programs.

**LEMMA 16.** *Let  $m, m'$  be equivalent Moore automata. Consider a sequence  $\sigma_1 \dots \sigma_n$  of environments.*

1. *There is a run  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  of  $\phi^{-1}(m)$  if and only if there is a run  $(l'_1, \sigma_1) \dots (l'_n, \sigma_n)$  of  $\phi^{-1}(m')$ .*
2. *If the above runs exist, they induce identical jump sequences and identical abstract execution sequences:*

$$\begin{aligned}
\mathcal{J}(l_1 \dots l_n) &= \mathcal{J}(l'_1 \dots l'_n) \\
\mathcal{A}(l_1 \dots l_n) &= \mathcal{A}(l'_1 \dots l'_n)
\end{aligned}$$

*Proof.* We proceed by induction on  $n$ . The base case is trivial. For the inductive step, assume  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  is a run of  $\phi^{-1}(m)$ . By induction, there is a run  $(l'_1, \sigma_1) \dots (l'_{n-1}, \sigma_{n-1})$  such that

$$\begin{aligned}
\mathcal{J}(l_1 \dots l_{n-1}) &= \mathcal{J}(l'_1 \dots l'_{n-1}) \\
\mathcal{A}(l_1 \dots l_{n-1}) &= \mathcal{A}(l'_1 \dots l'_{n-1})
\end{aligned} \tag{5}$$

Because  $(l_1, \sigma_1) \dots (l_n, \sigma_n)$  is a run,  $\llbracket \lfloor l_{n-1} \rfloor \rrbracket_B = (l_n, \sigma_n)$ . By (5) we have  $\mathcal{A}(l_{n-1}) = \mathcal{A}(l'_{n-1})$ . We invoke Lemma 13 and find an  $l'_n$  such that  $\llbracket \lfloor l'_{n-1} \rfloor \rrbracket_B = (l'_n, \sigma_n)$  and  $\mathcal{J}(l_{n-1} \ l_n) = \mathcal{J}(l'_{n-1} \ l'_n)$ . Hence there

is a run  $(l'_1, \sigma_1) \dots (l'_n, \sigma_n)$  of  $\phi^{-1}(m')$  and  $\mathcal{J}(l_1 \dots l_n) = \mathcal{J}(l'_1 \dots l'_n)$ . Using Corollary 15, we compute

$$\begin{aligned} \mathcal{A}(l_1 \dots l_n) &= \llbracket \phi(\phi^{-1}(m)) \rrbracket (\mathcal{J}(l_1 \dots l_n)) \\ &= \llbracket m \rrbracket (\mathcal{J}(l_1 \dots l_n)) \\ &= \llbracket m' \rrbracket (\mathcal{J}(l'_1 \dots l'_n)) \\ &= \llbracket \phi(\phi^{-1}(m')) \rrbracket (\mathcal{J}(l'_1 \dots l'_n)) \\ &= \mathcal{A}(l'_1 \dots l'_n). \end{aligned}$$

Finally, we have a proof of Theorem 4.

*Proof of Theorem 4.* Immediate from Lemma 16.

## 7. Discussion & Future Work

When one of our instrumented interpreters considers some basic block for transformation, it bases its decision on the particular execution path taken to reach that basic block. This accounts for the ease with which we have implemented otherwise complicated analyses and transformations: Reasoning about a single execution path is inherently simpler than reasoning about all possible execution paths.

However, this simplicity comes at a price. We are restricted to transformations that can be justified by inspecting only a single execution path. Consider *constant propagation* (Muchnick, 1997). Offhand, constant propagation seems an easy transformation to implement by the interpretive approach. Skip assignments  $x := c$  (where  $c$  is a constant), but do record  $x := c$  in a constant-propagation history; clear entries in the constant-propagation history when variables are reassigned; then try consulting the constant-propagation history before looking up variables in the store. Presto, constants are propagated. However, consider the effect of this implementation on the following fragment

```
if ... do
  x := 0;
else
  x := 1;
end;
<more code>
```

Here, there are no opportunities for constant propagation, because the value of  $x$  at the `<more code>` program point is not statically known. However, the suggested implementation would yield

```

if ... do
  <more code, specialized to x := 0>
else
  <more code, specialized to x := 1>
end;

```

This transformation is correct, but much more aggressive than constant propagation — it more resembles partial evaluation. Traditionally, constant propagation implementations check whether an assignment  $x := c$  is available in *all* predecessors. When using the interpretive approach, we can work with only a single path of execution, so the interpreter simply cannot inspect all the predecessors simultaneously. There seem to be no simple way to avoid this over-optimization. We expect that implementing copy propagation and dead-code elimination will prove similarly difficult. For the latter, there is the added complication that liveness is a property of the future, whereas our interpreters can only collect information about the past.

*Future Work* We see five major directions for future work.

1. Prove correctness of our implementations. Assuming correctness of the specializer, it is sufficient to prove our interpreters semantically equivalent. Such a proof would connect our results to ongoing work in provably correct compiler construction (Lacey et al., 2002; Lerner et al., 2003; Benton, 2004).
2. Explore more transformations. One challenge is to overcome the above-mentioned difficulties with constant propagation, copy propagation and dead-code elimination. Another is to investigate languages with pointers, requiring aliasing analysis.
3. Explore construction of optimizing compilers for domain-specific languages. Our specializer is standard except for rewinding, which is merely a variant of Moore automaton minimization, so we can reasonably hope to feasibly construct such optimizing compilers by specialization.
4. Integrate rewinding with specialization. Currently, we may produce exponentially large residual programs only to cut them back down with rewinding, incurring correspondingly exponential time consumption.
5. Formulate rewinding for functional languages, in particular higher-order functional languages.

## 8. Acknowledgments

The author wishes to thank Torben Mogensen for useful discussions. In addition, the author gratefully acknowledges detailed and constructive comments from Mads Sig Ager, Jakob Grue Simonsen, Lars Birkedal and the anonymous referees.

## References

- Benton, N.: 2004, ‘Simple relational correctness proofs for static analyses and program transformations’. In: *POPL ’04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA, pp. 14–25, ACM Press.
- Debois, S.: 2004, ‘Imperative program optimization by partial evaluation’. In: *PEPM ’04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. New York, NY, USA, pp. 113–122, ACM Press.
- Denning, P. J., J. B. Dennis, and J. E. Qualitz: 1978, *Machines, languages, and computation*. Prentice Hall.
- Glück, R. and J. Jørgensen: 1994, ‘Generating optimizing specializers’. In: *Proceedings of the 1994 IEEE International Conference On Computer Languages*. pp. 183–194, IEEE Computer Society Press.
- Glück, R. and J. Jørgensen: 1994, ‘Generating transformers for deforestation and supercompilation.’. In: *SAS ’94: Proceedings of the 1st international Static analysis symposium*, Vol. 864 of *Lecture Notes in Computer Science*. pp. 432–448.
- Gomard, C. K. and N. D. Jones: 1991, ‘Compiler generation by partial evaluation: a case study’. *Structured Programming* **12**, 123–144.
- Hopcroft, J. E. and J. D. Ullman: 1979, *Introduction to automata theory, languages and computation*. Addison–Wesley.
- Jones, N. D.: 2004, ‘Transformation by interpreter specialisation’. *Science of Computer Programming* **52**, 307–339.
- Jones, N. D., C. K. Gomard, and P. Sestoft: 1993, *Partial evaluation and automatic program generation*. Prentice Hall.
- Lacey, D., N. D. Jones, E. V. Wyk, and C. C. Frederiksen: 2002, ‘Proving correctness of compiler optimizations by temporal logic’. In: *POPL ’02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA, pp. 283–294, ACM Press.
- Lerner, S., T. Millstein, and C. Chambers: 2003, ‘Automatically proving the correctness of compiler optimizations’. In: *PLDI ’03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA, pp. 220–231, ACM Press.
- Muchnick, S. S.: 1997, *Advanced compiler design & implementation*. Morgan Kaufmann Publishers.
- Song, L., Y. Futamura, R. Glück, and Z. Hu: 2000, ‘Loop quasi-invariance code motion’. *IEICE Transactions on Information and Systems* **E83-D**(10), 1841–1850.
- Sperber, M. and P. Thiemann: 1996, ‘Realistic compilation by partial evaluation’. In: *PLDI ’96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*. pp. 206–214, ACM Press.

Thibault, S., C. Consel, J. L. Lawall, R. Marlet, and G. Muller: 2000, 'Static and dynamic program compilation by interpreter specialization'. *Higher-Order and Symbolic Computation* **13**(3), 161–178.

## Appendix

### A. Macros

```

1  # Determine if elem is a an element of list
   macro member (elem, list) :
       tmp := list;
4   while tmp != '() do
       if hd tmp = elem do
           return 'true;
       end;
8   tmp := tl tmp;
   end;
   return '();
end
12 # Assuming 'list' is associative and has key 'key', find the
    # corresponding value.
    macro lookup (key, list) :
16   tmp := list;
       while 'true do
           if hd (hd tmp) = key do
20             return tl (hd tmp);
           end;
           tmp := tl tmp;
       end;
       return '(bad lookup);
24 end

    # Reverse xs.
    macro reverse (xs) :
28   left := xs;
       right := '();
       while left != '() do
32         right := cons(hd left, right);
           left := tl left;
       end;
36   return right;
end

    # Return first argument of the expression of assignment.
40 macro param1Of (asgn) :
       return hd (tl (tl asgn));
end

44 # Assuming there is one, return second argument
    # of expression of assignment.
    macro param2Of (asgn) :
       return hd (tl (tl (tl asgn)));
48 end

    # Construct assignment 'id := op (arg1, arg2)'.
52 macro mkBinOpAsgn (id, op, param1, param2) :
       return cons (id, cons (op,
                               cons (param1,
                                     cons (param2, '()))));
end

```

```
56 # Return assignments target variable.
macro def (asgn) :
      return hd asgn;
60 end

# Return list of identifiers referenced by assignment.
macro use (asgn) :
64   ids := '();
      if 'id = hd (param10f (asgn)) do
          ids := cons (tl (param10f (asgn)), ids);
68   end;
      has2Args := tl (tl (tl asgn));
      if has2Args do
72   if 'id = hd (param20f (asgn)) do
          ids := cons (tl (param20f (asgn)), ids);
          end;
        end;
76   return ids;
      end
end
```

*Address for Offprints:* IT University of Copenhagen  
Rued Langgaardsvej 7  
2300 Copenhagen S  
Denmark