

Imperative Program Optimization by Partial Evaluation

Søren Debois
Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1
DK-2100 Copenhagen
Denmark
debois@diku.dk

ABSTRACT

We implement strength reduction and loop-invariant code motion by specializing instrumented interpreters; we define a novel program transformation that uses bisimulation to identify and remove code duplication in residual programs; and we discover that some simple classical optimizations, notably constant-propagation, seemingly do not lend themselves to implementation by specialization of instrumented interpreters.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Optimization*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*Partial evaluation*

General Terms

Experimentation

Keywords

Strength reduction, loop-invariant code motion, code duplication, bisimulation

1. INTRODUCTION

In this paper, we describe how to implement the imperative program transformations strength reduction and loop-invariant code motion [12] by the interpretive approach [4] — that is, performing said program transformations by specializing instrumented interpreters. We define a novel program transformation we call *rewinding*, which uses bisimulation [11] to identify and remove code duplication. Finally, we discover that constant propagation [12] does not lend itself to the interpretive approach.

Our contributions are significant for at least the following reasons. First, we provide a simple alternative implementation for well-known, non-trivial imperative program transformations. Second, we provide insight into the strengths

and weaknesses of the interpretive approach in an imperative setting. Finally, we provide the rewinding program transformation for undoing code duplication and a theoretical means of reasoning about code duplication; both are generally applicable.

How does the interpretive approach apply in an imperative setting? We instrument a simple interpreter to collect statically available information as it runs, thus establishing a *history*, which is subsequently used to justify optimizations. For example, an interpreter may perform loop-invariant code motion on an assignment $x := \langle \text{exp} \rangle$ by noting whether a particular execution of the assignment would change the value of x . If it would not, the interpreter simply skips the assignment. The interpreter uses the 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, applying a polyvariant specializer to a history-collecting interpreter is likely to produce code duplication, in the same way that dead static variables do [6, 8, 7]. This duplication is intrinsic to our methods; we cannot prevent it. Instead, we apply our rewinding transformation to the residual program, thereby removing duplication.

We believe that ours is the first substantial application of the interpretive approach to an imperative language ([15] contains an incidental application to a domain-specific imperative language). For functional languages, the interpretive approach has been used to implement constant propagation, higher-order removal [14], super-compilation [4, 5] and deforestation [5].

This paper is organized as follows: In Section 2, we introduce the Flowchart language, a simple interpreter for it, and our specializer; in Section 3, we describe how to implement loop-invariant code motion; in Section 4, we describe how to implement strength reduction; in Section 5, we discuss code duplication; in Section 6, we define bisimulation and rewinding; and in Section 7, we discuss our methods and point out directions for future work.

2. PRELIMINARIES

We use a variant of the Flowchart language of [6, 8]. Syntax for the language is given in Figure 1. 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, a conditional jump or program termination. (See Figure 4 for a simple example program.)

We assume throughout that there are no jumps to undefined labels, that the label of each basic block is unique,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'04, August 24–26, 2004, Verona, Italy.

Copyright 2004 ACM 1-58113-835-0/04/0008 ...\$5.00.

```

Program ::= Input Block+
Input  ::= read Id*
Block  ::= Label Asgns Jump
Jump   ::= goto Label
        | if Param then Label else Label
        | return Param
Asgns  ::= Asgn*
Asgn   ::= Id := Expr
Expr   ::= Op Param+
        | Param
Param  ::= Id
        | ' SExp
SExp   ::= atom
        | ( SExp* )
Op     ::= hd | tl | cons | + | - | = | < | ...

```

Figure 1: The Flowchart Language

and that each operator application has the correct number of parameters. Values for the Flowchart language are the S-expressions of LISP; that is, values are either atoms or pairs of values, the former being either strings or numbers. Semantics for Flowchart is as expected, execution commencing at the first basic block, the *entry* block.

Definition 1. Let P be a flowchart program, and let l be a label of P . We shall write $[l]^P$ to denote the basic block of P with label l , occasionally dropping the P if it is clear from the context. Let \mathbb{D} be some non-empty set with $\text{False} \in \mathbb{D}$. An *environment* is a finite partial map $\sigma : \text{Id} \rightarrow \mathbb{D}$. Denote by Σ the set of such σ . Assume total functions $\mathcal{A}[\cdot] : \text{Asgns} \rightarrow \Sigma \rightarrow \Sigma$, executing assignments; and $\mathcal{P}[\cdot] : \text{Param} \rightarrow \Sigma \rightarrow \mathbb{D}$, evaluating parameters. (The simple interpreter of Figure 3 implements \mathcal{A} and \mathcal{P} in the natural way.) Define a function $\mathcal{J}[\cdot] : \text{Jump} \rightarrow \Sigma \rightarrow \text{Label} \times \Sigma$ by

$$\begin{aligned} \mathcal{J}[\text{goto } l] \sigma &= (l, \sigma) \\ \mathcal{J}[\text{if } p \text{ then } l_1 \text{ else } l_2] \sigma &= \begin{cases} (l_2, \sigma) & \text{if } \mathcal{P}[p] \sigma = \text{False} \\ (l_1, \sigma) & \text{otherwise} \end{cases} \\ \mathcal{J}[\text{return } p] \sigma &= (\bullet, \sigma[Y \leftarrow \mathcal{P}[p] \sigma]), \end{aligned}$$

where \bullet is different from all labels and Y is different from all variable names. Define a total function $\mathcal{B}[\cdot] : \text{Block} \rightarrow \Sigma \rightarrow \text{Label} \times \Sigma$ by $\mathcal{B}[l \text{ Asgns Jump}] \sigma = \mathcal{J}[\text{Jump}] (\mathcal{A}[\text{Asgns}] \sigma)$. A *run* of length n of P on environment σ_1 is a sequence $(l_1, \sigma_1) \dots (l_n, \sigma_n)$ such that $l_i \neq \bullet$ for $1 \leq i < n$, $(l_i, \sigma_i) = \mathcal{B}[[l_{i-1}]^P] \sigma_{i-1}$ for $1 < i \leq n$, and l_1 is the label of the entry block of P . Finally, we say that P *terminates on* σ_1 *with value* d if there exists a run $(l_1, \sigma_1) \dots (l_n, \sigma_n)$ such that $l_n = \bullet$ and $\sigma_n(Y) = d$.

Writing programs in Flowchart is unendurably tedious, so we allow as syntactic sugar **if**, **while**, and **case**; compound expressions; infix **cons** operator **::**; short-circuiting boolean expressions and non-recursive functions (macros).

We implement Flowchart interpreters in Flowchart itself, extended with an indexed store construct, which is handy for implementing symbol tables. The extended language, Flowchart⁺, have the two extra constructs shown in Figure 2: A new assignment form allows the assignment of a value to a store location, and a new operator allows the re-

```

:
:
Asgn ::= ...
      | [ Param ] := Param
:
Op   ::= ... | load

```

Figure 2: The Flowchart⁺ Language

trieval of a value from a store location. We use ‘ $[Param]$ ’ as syntactic sugar for ‘load $Param$ ’.

To carry out our experiments, we have implemented a rudimentary polyvariant, transition-compressing, program-point specializer from Flowchart⁺ to Flowchart, as described in [8]. The residual programs exhibited in this paper are all actual output of this specializer.

A Flowchart-interpreter written in Flowchart⁺ 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 standard macros 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 18–64). The assignments of the current block are executed (lines 22–48) by evaluating the parameters (lines 26–33), and applying the appropriate operator to the values of the parameters (lines 34–46). Finally, the jump of the basic block is executed (lines 49–64).

A note on terminology: We will frequently need to state that “ P is the result of specializing interpreter **int** w.r.t. program P' ”. This quickly becomes unwieldy, so we shall say simply P' *transformed by int yields* P .

3. LOOP-INVARIANT CODE MOTION

Loop-invariant code motion [12] 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 $\mathbf{t} := 2 * \mathbf{n}$ in line 5 is recomputed on each iteration of the loop, even though the value of \mathbf{n} never changes. We shall see how transforming **fac** by an instrumented interpreter may yield the program in Figure 7.

Observe that executing an assignment $\mathbf{x} := \langle \text{exp} \rangle$ cannot change the value of \mathbf{x} if (a) the previous assignment to \mathbf{x} were also $\mathbf{x} := \langle \text{exp} \rangle$ and (b) the variables used in $\langle \text{exp} \rangle$ were not updated in the meantime. If these conditions are satisfied, we say that the assignment $\mathbf{x} := \langle \text{exp} \rangle$ is *available*.

Maintaining a list of available assignments is easy in an interpreter: Every time an assignment $\mathbf{x} := \langle \text{exp} \rangle$ has been executed, we remove all assignments that use or define \mathbf{x} from the available assignments list. Then we add the assignment $\mathbf{x} := \langle \text{exp} \rangle$ to the list, provided $\langle \text{exp} \rangle$ does not reference \mathbf{x} (thus, we avoid self-assignment). The available assignments are a subset of the assignments of the input program, so the available assignments list is of bounded static variation [8]. Thus, recording available assignments does not introduce extra specialization-time non-termination in the interpreter.

```

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;
   formal := tl (hd pgm);
   while formal != '() do
12    [hd formal] := hd data;
       formal := tl formal;
       data := tl data;
   end;
16  bbs := hd (tl pgm);
   next := hd (hd bbs);
   exec-bb:
   bb := lookup (next, bbs);
   asgns := hd bb;
   jump := tl bb;
   while asgns != '() do
24    asgn := hd asgns; asgns := tl asgns;
       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
32         param2 := eval-param (hd params);
           end;
       case op of
36         'hd: [id] := hd param1;
           'tl: [id] := tl param1;
           'not: [id] := ! param1;
           'base: [id] := param1;
           'eq: [id] := param1 = param2;
           'cons: [id] := cons (param1,param2);
           'add: [id] := param1 + param2;
           'mult: [id] := param1 * param2;
           'less: [id] := param1 < param2;
44         default:
           return '(bad operator);
       end;
   next-asgn:
48  end;
   case hd jump of
   'goto:
52  next := tl jump;
   'return:
   return eval-param (tl jump);
   'if:
56  param := hd (tl jump);
       labtrue := hd (tl (tl jump));
       labfalse := tl (tl (tl jump));
       if eval-param (param) do
60         next := labtrue;
       else
       next := labfalse;
       end;
64  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.

A code fragment maintaining the available assignments list is given in Figure 5. The fragment is inserted immedi-

```

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

```

Figure 5: Implementation of available assignments analysis.

ately after the code for execution of an assignment — that is, between lines 46 and 47 in Figure 3. Suppose an assignment $id := \langle \text{exp} \rangle$ has just been executed. 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, respectively, list reversal, list membership, the list of variables referenced in an assignment and the variable defined by an assignment.

The code fragment in Figure 6 skips execution of superfluous assignments. It is inserted immediately before the code for assignment execution — that is, between lines 26 and 27 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
4     asgn' := hd tmp;
       if asgn = asgn' do
           goto next-asgn;
       end;
       tmp := tl tmp;
8  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 test part of the loop

```

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

```

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

(lines 5–7 of the original program in Figure 4) has been unrolled once in the residual program in Figure 7: on the first iteration (lines 6–8), $t := 2 * n$ is computed; on subsequent iterations (lines 12–13), it is skipped.

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 unrolled the loop, in the process removing the redundant computation from the second iteration onwards. We exploit that unrolling the loop reduces loop-invariant code motion to a trivial case of common-subexpression elimination.

4. STRENGTH REDUCTION

Consider the program `div` of Figure 8. This program computes the function

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

where $\lceil x \rceil$ is the least integer greater than or equal to x . Since the variable j will assume the values $0, 3d, 6d, 9d, \dots$,

```

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

Figure 8: Program `div`.

we can replace line 9 with $j := j + (3 * d)$. The expression $3 * d$ is loop-invariant, so if we compute it outside of the loop and store the result in some fresh variable, say, $v0 := 3 * d$, we can change line 9 to $j := j + v0$. This may be an optimization if addition is faster than multiplication. Automatically converting multiplications to additions is called *strength reduction* [12].

In the above example, we replace the expression $i * d$ with $j + (3 * d)$. What are the values of these variables when we are about to evaluate $i * d$? Using i , d and j to denote the values of i , d and j , respectively, and using i' to denote the previous value of i , we find

$$\begin{aligned} j &= i'd \\ i &= i' + 3 \end{aligned} \quad (1)$$

Thus

$$id = (i' + 3)d = i'd + 3d = j + 3d.$$

Since $i * d$ evaluates to id , we may replace an expression $i * d$ with $j + (3 * d)$, provided the *values* of i , j and d are related by some value i' as specified in (1). Obviously, we can generalize the “3” to an arbitrary constant c . In this case

$$\begin{aligned} j &= i'd \\ i &= i' + c \end{aligned} \quad (2)$$

and

$$id = (i' + c)d = i'd + cd = j + cd.$$

Thus, for any constant c , we may replace an expression $i * d$ with $j + (c * d)$, provided the values i , d and j are related by some i' as specified by (2)

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 strength reduce the `div` program. Consider a sequence of assignments

$$j := i * d, \dots, i := i + c, \dots, j := i * d. \quad (3)$$

Any such sequence produces values satisfying Condition (2), provided that d 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. We keep the reaching definitions of bounded static variation 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 then performed by the interpreter replacing the statement $j := i * d$ with the two statements $v0 := c * d$ and $j := j + v0$, where $v0$ is some fresh variable. Loop-invariant hoisting can then move the former statement out of the loop.

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

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

This sequence does not have the form (3), so there will be no strength reduction.

Strength reduction is implemented in Figure 9 (collecting reaching definitions) and 10 (recognizing sequences in the form (3) and actually performing strength reduction). The former is to be inserted along with the available assignments computation (Figure 5) between lines 46 and 47 of the simple interpreter (Figure 3), the latter is to be inserted along with the hoisting implementation (Figure 6) between lines 26 and 27.

```

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 9: Implementation of reaching definitions analysis.

Although it may appear intimidating, the implementation of strength reduction in Figure 10 derives its complexity

```

1   if hd exp = 'mult &&
    hd (param1Of (asgn)) = 'id &&
    hd (param2Of (asgn)) = 'id
4   do
    j := def (asgn);
    i := tl (param1Of (asgn));
    d := tl (param2Of (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) = param1Of (asgn') &&
        'quote = hd (param2Of (asgn'))
        do
16       c := tl (param2Of (asgn'));
          goto found-incr;
        else
20       goto abort-strength-reduction;
        end;
      end;
      if j = def (asgn') ||
        d = def (asgn')
24     then 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;
32         if i = def (asgn') ||
            d = def (asgn')
          then abort-strength-reduction;
        end;
36     goto abort-strength-reduction;
do-strength-reduction:
    tmp := next :: asgns;
    mult := mkBinOpAsgn (tmp, 'mult,
                        'quote :: c,
                        'id :: d);
    asgn' := mkBinOpAsgn (j, 'add,
                        'id :: j,
                        'id :: tmp);
44     asgns := mult :: asgn' :: asgns;
        revision := mult :: asgn :: '();
        goto next-asgn;
48 abort-strength-reduction:
    end;

```

Figure 10: Implementation of strength reduction.

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 have the form (3).

Note the construction of a name for the necessary temporary variable in line 38: 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 11. Observe 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   read data;
L0:
    n := hd data;
    data := tl data;
    d := hd data;
    i := 0;
    j := 0;
8   t := j < n;
    if t then L1 else L2;
L1:
    i := i + 3;
    j := i * d;
    t := j < n;
12    if t then L3 else L2;
L2:
16    return i;
L3:
    i := i + 3;
    v0 := 3 * d;
    j := j + v0;
20    t := j < n;
    if t then L9 else L2;
L9:
24    i := i + 3;
    j := j + v0;
    t := j < n;
    if t then L9 else L2;

```

Figure 11: Transformation of `div` (Figure 8) by the strength-reducing interpreter.

5. CODE DUPLICATION

Consider the program `count` of Figure 12. Given an integer n and an integer list l , this program computes the number of elements of l greater than n .

```

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

```

Figure 12: Program `count`.

Transforming the `count` program by the hoisting interpreter, but omitting post-specialization rewinding, yields the program in Figures 13 and 14. The latter figure represents the program graphically: solid edges corresponds to `then` branches, dashed edges to `else` branches.

Obviously, the transformed program contains redundant blocks. Comparing it to the original, we see the block `L0` corresponds to the concatenation of the blocks `start` and `loop`, modulo labels and argument-reading artifacts of the interpreter. This is as expected: Transition compression dictates the concatenation of blocks connected by an unconditional jump.

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.

The blocks `L3` and `L4` both end with code corresponding to `loop`, so we would expect `L3` and `L4` to branch to the

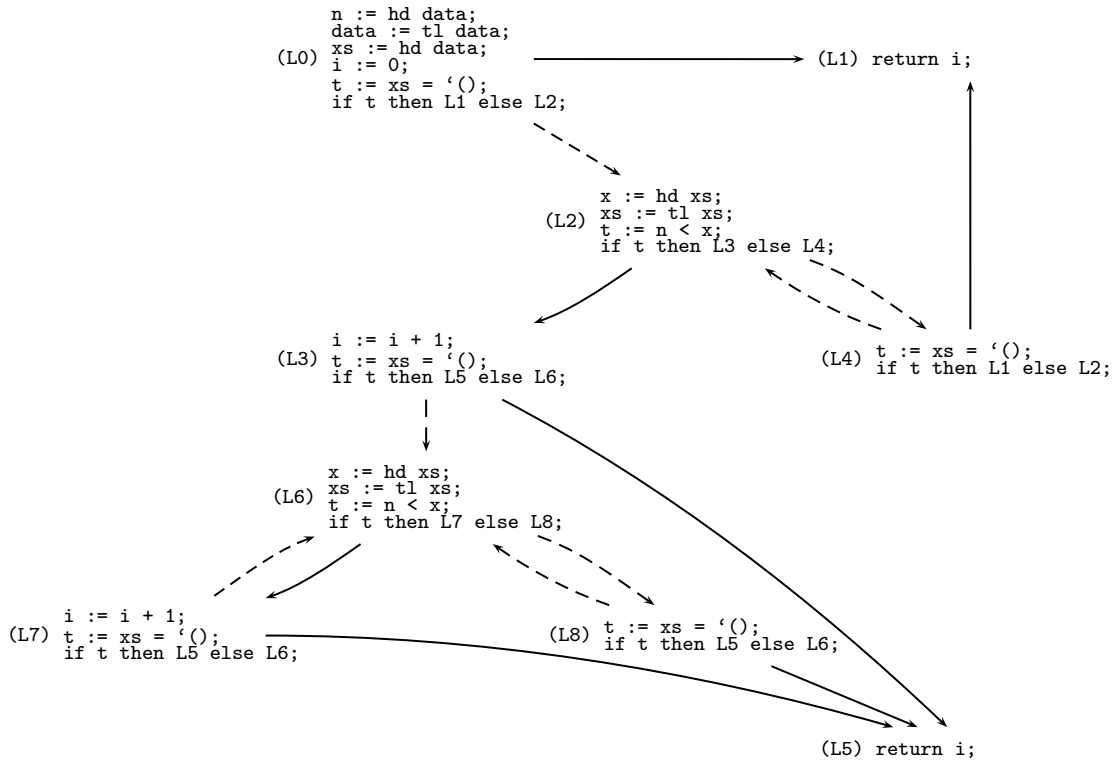


Figure 14: Flowgraph representation of the program of Figure 13.

```

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

```

Figure 13: Transformation of count (Figure 12) by the hoisting interpreter, omitting post-specialization rewinding.

same continuations. Unfortunately, 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.

Clearly, 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? Observe that the available assignments at the `loop` block depend on the path taken to reach `loop`. First, if one arrives via `start`, the assignment `i := 0` is available. Second, if one arrives via `inc`, no assignment to `i` is available, because of the self-assignment `i := i + 1` in `inc`. Finally, if one arrives via `body`, the available assignments for `i` depends on the path taken to reach the `body` block, since the `body` block does not itself change availability of `i`.

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). Thus, any history-collecting interpreter is likely to introduce code duplication.

This duplication has the flavor of the dead static variable problem [6, 7, 8]. 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.

To do so, we must find some notion of equivalence that identifies redundant basic blocks. Obviously, this notion of equivalence must deem L1 and L5 equivalent. It also appears that L2 and L6 should be equivalent, as should (L3, L7) and (L4, L8). The blocks in each of these pairs 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 is essentially a variation on *bisimulation equivalence* [11]. Thus, we undo code duplication by rewinding the residual program into a minimal bisimulation equivalent program. In Section 6, we define bisimulation on programs and the rewinding transformation; for now, we simply postulate that in the program of Figures 13 and 14, the blocks in the pairs (L1, L5), (L2, L6), (L3, L7) and (L4, L8) are bisimulation equivalent, respectively. Applying the rewinding transformation yields the program in Figure 15; note how the blocks L1 – L4 have been removed.

```

1  read data;
   L0:
4      n := hd data;
       data := tl data;
       xs := hd data;
       i := 0;
       t := xs = '();
8      if t then L5 else L6;
   L5:
       return i;
   L6:
12     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 15: Transformation of count (Figure 12) by the hoisting interpreter, including post-specialization rewinding.

6. BISIMULATION & REWINDING

To define rewinding, we need a notion of bisimulation on Flowchart programs. But first, notation.

Definition 2. Let P be a program. We use P to denote also the set of basic blocks of P . Let b and b' be basic blocks with $b = l \text{ Asgns } \text{Jump}$ and $b' = l' \text{ Asgns}' \text{ Jump}'$. We define $b \equiv b'$ iff $\text{Asgns} = \text{Asgns}'$ and one of the following holds:

1. $\text{Jump} = \text{Jump}'$,
2. for some l_1 and l'_1 , $\text{Jump} = \text{goto } l_1$ and $\text{Jump}' = \text{goto } l'_1$,
3. for some l_1, l'_1, l_2 and l'_2 , $\text{Jump} = \text{if } p \text{ then } l_1 \text{ else } l_2$ and $\text{Jump}' = \text{if } p \text{ then } l'_1 \text{ else } l'_2$.

(Notice that \equiv is an equivalence relation.) Recall from Definition 1 that for a label l of P , we write $[l]^P$ to denote the

basic block of P with label l , dropping the P if clear from the context. We define the *successors of b* as a possibly empty sequence of basic blocks

$$\text{succ}(b) = \begin{cases} [l_1], [l_2] & \text{if } \text{Jump} = \text{if } p \text{ then } l_1 \text{ else } l_2 \\ [l] & \text{if } \text{Jump} = \text{goto } l \\ \epsilon & \text{otherwise} \end{cases}$$

(The program P will be clear from the context.) We denote by $\text{succ}_j(b)$ the j 'th successor of b , and by $|\text{succ}(b)|$ the length of the sequence $\text{succ}(b)$ — that is, the number of successors to b .

Definition 3. Let P, P' be flowchart programs. We define inductively binary relations $B_i \subseteq P \times P'$ on the basic blocks of P and P' by $(b, b') \in B_0$ iff $b \equiv b'$, and $(b, b') \in B_{i+1}$ iff both

1. $b \equiv b'$, and
2. $(\text{succ}_j(b), \text{succ}_j(b')) \in B_i$ for each $1 \leq j \leq |\text{succ}(b)|$.

We say that basic blocks $b \in P$ and $b' \in P'$ are *bisimulation equivalent* and write $b \sim b'$ iff

$$(b, b') \in \bigcap_{i=0}^{\infty} B_i.$$

We say that *programs P, P' are bisimulation equivalent* and write $P \sim P'$ iff the entry blocks of P and P' are bisimulation equivalent. (Notice that a bisimulation $B \subseteq P \times P$ on a single program P is an equivalence relation.)

Informally, $(b, b') \in B_i$ means that b and b' cannot be distinguished in i computation steps (jumps) or less. Thus b and b' are bisimulation equivalent if they cannot be distinguished by any finite number of computation steps.

It is straightforward to construct an algorithm that given a program P computes the bisimulation on $P \times P$ — we give one in Appendix A. (Contrary to the standard formulations of bisimulation on graphs or Kripke-structures, the successors in Flowchart programs are ordered. Thus, the standard algorithm for computing bisimulations [13] does not apply.)

We prove that bisimulation equivalence entails semantic equivalence. We will need the following Lemma, which is easily proved by induction on n :

LEMMA 1. *For any program P , environment σ and $n \geq 1$, there exists at most one run of length n .*

Definition 4. For any program P and any environment σ , we define the function

$$[[P]]\sigma_1 = \begin{cases} d & \text{if } P \text{ terminates on } \sigma \text{ with value } d \\ \perp & \text{otherwise} \end{cases}$$

(This is well-defined by Lemma 1.)

THEOREM 1. *Let P_1 and P_2 be bisimulation equivalent programs, and let $n \geq 1$ be some natural number. For any sequence $\sigma_1 \dots \sigma_n$, the following holds:*

1. *There exists a run $(l_1^1, \sigma_1) \dots (l_n^1, \sigma_n)$ of P_1 on σ_1 iff there exists a run $(l_1^2, \sigma_1) \dots (l_n^2, \sigma_n)$ of P_2 on σ_1 .*
2. *If the above runs exist, then $[l_i^1]^{P_1} \sim [l_i^2]^{P_2}$ for $1 \leq i \leq n$.*

PROOF. By induction on n . For $n = 1$, consider some singleton sequence σ_1 . For (1), we trivially have runs (l_1^1, σ_1) and (l_1^2, σ_1) , and for (2), we have by definition $\llbracket l_1^1 \rrbracket^{P_1} \sim \llbracket l_1^2 \rrbracket^{P_2}$. For some $n > 1$ and some sequence $\sigma_1 \dots \sigma_n$, assume that there exists a run $(l_1^1, \sigma_1) \dots (l_n^1, \sigma_n)$ (the other case is symmetric). By the induction hypothesis, there exists a run $(l_1^2, \sigma_1) \dots (l_{n-1}^2, \sigma_{n-1})$ with the desired properties. In particular,

$$\llbracket l_{n-1}^1 \rrbracket^{P_1} \sim \llbracket l_{n-1}^2 \rrbracket^{P_2}. \quad (4)$$

Assume $\llbracket l_{n-1}^1 \rrbracket^{P_1} = l_{n-1}^1 \text{ Asgns}_1 \text{ Jump}_1$ and $\llbracket l_{n-1}^2 \rrbracket^{P_2} = l_{n-1}^2 \text{ Asgns}_2 \text{ Jump}_2$, and define $\sigma = \mathcal{A}[\llbracket \text{Asgns}_1 \rrbracket \sigma_{n-1}]$. By (4) we have $\text{Asgns}_1 = \text{Asgns}_2$, hence $\sigma = \mathcal{A}[\llbracket \text{Asgns}_1 \rrbracket \sigma_{n-1}] = \mathcal{A}[\llbracket \text{Asgns}_2 \rrbracket \sigma_{n-1}]$. But then $\mathcal{B}[\llbracket l_{n-1}^1 \text{ Asgns}_1 \text{ Jump}_1 \rrbracket \sigma_{n-1}] = \mathcal{J}[\llbracket \text{Jump}_1 \rrbracket](\mathcal{A}[\llbracket \text{Asgns}_1 \rrbracket \sigma_{n-1}]) = \mathcal{J}[\llbracket \text{Jump}_1 \rrbracket] \sigma$, and similarly $\mathcal{B}[\llbracket l_{n-1}^2 \text{ Asgns}_2 \text{ Jump}_2 \rrbracket \sigma_{n-1}] = \mathcal{J}[\llbracket \text{Jump}_2 \rrbracket] \sigma$. We proceed by examining the possible forms of Jump_1 .

$\text{Jump}_1 = \text{goto } l^1$. By the definition of $\mathcal{J}[\cdot]$ and Lemma 1 $\mathcal{J}[\llbracket \text{goto } l^1 \rrbracket] \sigma = (l^1, \sigma) = (l_n^1, \sigma_n)$, hence $l^1 = l_n^1$ and $\sigma = \sigma_n$. By (4), $\text{Jump}_2 = \text{goto } l^2$ for some l^2 , so $\mathcal{J}[\llbracket \text{goto } l^2 \rrbracket] \sigma = (l^2, \sigma) = (l_n^2, \sigma_n)$. Thus, $(l_1^1, \sigma_1) \dots (l_{n-1}^1, \sigma_{n-1})(l_n^1, \sigma_n)$ is indeed a run of length n of P_2 on σ_1 . By (4) and the definition of bisimilarity, $\llbracket l_n^1 \rrbracket^{P_1} \sim \llbracket l_n^2 \rrbracket^{P_2}$.

$\text{Jump}_1 = \text{if } p \text{ then } l'_1 \text{ else } l''_1$. By definition of $\mathcal{J}[\cdot]$ and Lemma 1, $\mathcal{J}[\llbracket \text{if } p \text{ then } l'_1 \text{ else } l''_1 \rrbracket] \sigma = (l_n, \sigma)$. By (4), we must have $\text{Jump}_2 = \text{if } p \text{ then } l'_2 \text{ else } l''_2$, hence

$$\mathcal{J}[\llbracket \text{Jump}_2 \rrbracket] \sigma = \mathcal{J}[\llbracket \text{if } p \text{ then } l'_2 \text{ else } l''_2 \rrbracket] \sigma = (l, \sigma),$$

with $l = l'_2$ or $l = l''_2$. Thus, $(l_1^2, \sigma_1) \dots (l_{n-1}^2, \sigma_{n-1})(l, \sigma_n)$ is indeed a run of length n of P_2 on σ_1 . By (4), we must have both $\llbracket l'_1 \rrbracket^{P_1} \sim \llbracket l'_2 \rrbracket^{P_2}$ and $\llbracket l''_1 \rrbracket^{P_1} \sim \llbracket l''_2 \rrbracket^{P_2}$, hence, whatever the value of $\mathcal{P}[\llbracket p \rrbracket] \sigma$, $\llbracket l_n^1 \rrbracket^{P_1} \sim \llbracket l_n^2 \rrbracket^{P_2}$.

Otherwise. By (4), $\text{Jump}_2 = \text{Jump}_1$. \square

COROLLARY 1. For any programs P_1 and P_2 , if $P_1 \sim P_2$ then $\llbracket P_1 \rrbracket = \llbracket P_2 \rrbracket$.

PROOF. Immediate from Lemma 1 and Theorem 1. \square

The algorithm in Figure 16 implements rewinding: finding minimal bisimulation equivalent programs. It is easy to see

Initialization Input a program P . Find the bisimulation B on $P \times P$.

General step For each non-trivial equivalence class $C \subseteq B$, do the following: choose a representative $c \in C$, such that if the entry block belongs to C then c is the entry block; remove from P all blocks $c' \neq c$ where $c' \in C$; and convert all jumps in P to such c' into jumps to c .

Output the modified P .

Figure 16: The rewinding transformation

that the general step of the algorithm preserves bisimilarity. Hence, correctness follows from Corollary 1.

THEOREM 2 (CORRECTNESS OF REWINDING). Let P be a program and P' the output of rewinding (Figure 16) when applied to P . Then $\llbracket P \rrbracket = \llbracket P' \rrbracket$.

The reader may wonder if rewinding is somehow related to *folding* in the sense of Darlington & Burstall [2]. (The reader not wondering so may safely skip to the next section.) We shall see that there is a certain resemblance. However, whereas folding can be unsafe, rewinding never is, and it appears that rewinding is in some cases more general than folding.

What does folding mean in Flowchart, which has no functions? Consider a first-order functional language with all function calls in the tail-position and with no nested conditionals. In this language, folding has an obvious meaning, and translating programs of this language into Flowchart is easy. We find a meaning for folding in Flowchart by translating the meaning of folding in this functional language.

For concreteness, consider this (admittedly rather contrived) functional program:

```
start (n) = if n then f(n) else g(n)
f (n)     = n+1
g (n)     = n+1
```

Translating to Flowchart, we get

```
read n;
start:  if n then f else g;
f:      t := n + 1;
        return t;
g:      t := n + 1;
        return t;
```

In general, folding is the transformation that replaces a substitution instance of a function body with the corresponding function application. We restrict ourselves to a particularly simple case of folding: replacing an entire function body with a call to some function having precisely the same body. For example, in the functional program above, we may fold the definition of f by g , obtaining

```
start (n) = if n then f(n) else g(n)
f (n)     = g(n)
g (n)     = n+1
```

Translating this program to Flowchart, we get

```
read n;
start:  if n then f else g;
f:      goto g;
g:      t := n + 1;
        return t;
```

Thus, it appears that in Flowchart, folding should mean replacing the body of a basic block with a jump to some basic block having precisely the same body.

It is well-known that folding indiscriminately may introduce non-termination: Just fold the definition of a function with the function itself. For example, we can fold the definition of g by g , obtaining $g(n) = g(n)$. Transferring this example to Flowchart, we see that we may fold the body of g by g , obtaining $g: \text{goto } g;$.

Rewinding cannot do such disruptive folding. First, it is impossible for rewinding to modify basic blocks, except for changing jump targets and throwing away unreachable blocks. Hence, rewinding cannot replace the two statements $t := n + 1; \text{return } t;$ in the g block with $\text{goto } g;$. Second, even if a block $l: \text{goto } l$ were present in the program, rewinding could not mistakenly identify l and g , as the two blocks are not bisimulation equivalent: They contain different assignments.

When folding does preserve semantics — i.e., when we replace the body of a basic block with a jump to some *other* basic block — it resembles rewinding. To wit, rewinding the original Flowchart program, we get

```

read n;
start:
  if n then g else g;
g:
  t := n + 1;
  return n;

```

Note how this closely resembles the result of folding the block `f` by `g` in the above example. In fact, rewinding appears, in this example, to be folding followed by transition compression. (Transition compression can be thought of as the Flowchart equivalent of a restricted “unfold” operation. Thus, in this example, we may think of rewinding as semantics preserving folding followed by unfolding.) However, rewinding appears to be more general than semantics preserving folding: Whereas folding in a sense identifies identical basic blocks, rewinding identifies bisimulation equivalent basic blocks. In the `count` program of Figures 13 and 14, rewinding identifies (among others) the blocks `L2` and `L6`, since they are bisimulation equivalent. However, folding cannot identify these blocks, as they are not identical — they contain the different branch targets `L3` and `L4` vs. `L7` and `L8`. For this reason, it appears that no sequence of fold/unfold transformations can convert the program in Figures 13 and 14 to the one in Figure 15.

7. DISCUSSION & FUTURE WORK

When one of our interpreters considers some basic block for optimization, it bases its decision on the particular execution path taken to reach said basic block. This accounts for the ease with which we have implemented otherwise complicated analysis and transformations: Reasoning about a single execution path is inherently simpler than reasoning about an entire flowgraph.

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* [12]. 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 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, since 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.

These limitations hint at directions for future work: Somehow overcoming the interpreter’s inherent limitation to a single execution path would enable us to express transformations currently out of reach. A different direction is further studying the rewinding transformation, in particular finding out whether it can somehow be integrated with specialization. Currently, we may produce enormous residual programs only to cut them back down with rewinding, incurring correspondingly enormous time consumption.

Our results have potential applications in the construction of a provably correct imperative program optimizer [9, 10, 1]: To prove our transformation implementations correct, it would be sufficient to prove the augmented interpreters semantically equivalent to a definitional interpreter, assuming correctness of the specializer.

Also, the simplicity of our implementations suggests that the interpretive approach may feasibly be used to construct optimizing compilers for domain-specific languages. Thus, our results complement recent findings [15] that compiling domain-specific languages by specialization can yield target programs performing comparably to hand-coded low-level programs.

Finally, the code duplication we experience is not exclusive to our interpreters. Any program that maintains a history of its own execution may incur code duplication. Thus, the rewinding transformation may apply generally to problems of code duplication in partial evaluation.

8. ACKNOWLEDGEMENTS

The author gratefully acknowledges detailed and constructive comments from Jakob Grue Simonsen, Neil Jones, Peter Sestoft, and the anonymous referees.

9. REFERENCES

- [1] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM Symposium On Principles of Programming Languages (POPL04)*, 2004.
- [2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
- [3] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.
- [4] R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE International Conference On Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.
- [5] R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*, volume 864 of

Lecture Notes in Computer Science, pages 432–448. Springer-Verlag, 1994.

- [6] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.
- [7] N. D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, pages 1–40, 2004. accepted for publication.
- [8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.
- [9] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29Th Annual ACM Symposium On Principles of Programming Languages (POPL02)*, pages 283–294. ACM Press, 2002.
- [10] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *ACM SIGPLAN Conference On Programming Language Design and Implementation (PLDI 2003)*, pages 220–231. ACM Press, 2003.
- [11] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.
- [12] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.
- [13] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal On Computing*, 16(6):973–989, Dec. 1987.
- [14] M. Sperber and P. Thiemann. Realistic compilation by partial evaluation. In *ACM SIGPLAN '96 Conference On Programming Language Design and Implementation, Philadelphia, Pennsylvania, May 1996 (SIGPLAN Notices, Vol. 31, No. 5)*, pages 206–214. New York: ACM, 1996.
- [15] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

APPENDIX

A. COMPUTING BISIMULATIONS

The algorithm in Figure 17 computes bisimulations on Flowchart programs. The algorithm combines ideas from both [13] and [3].

Initialization Input a program P . Let $C_0 = \{(b, b') \in P \times P \mid b \equiv b'\}$, and set $i = 0$.

General step Number the equivalence classes in C_i consecutively from 1. Assign to each basic block b the number $\alpha(b)$ of the equivalence class containing b . For each basic block b construct a *signature* $s(b) = (\alpha(b_1), \dots, \alpha(b_n))$ where b_1, \dots, b_n are the successors of b .

Let C_{i+1} be the coarsest equivalence relation such that $C_{i+1} \subseteq C_i$ and $(b_1, b_2) \in C_{i+1}$ implies $s(b_1) = s(b_2)$. If $C_{i+1} = C_i$ output C_i , otherwise set i to $i + 1$ and repeat the general step.

Figure 17: An algorithm computing bisimulations.

To prove correctness, we need the following lemma:

LEMMA 2. For any program P , the relations B_i of Definition 3 satisfy $B_{i+1} \subseteq B_i$ for $i \geq 0$.

PROOF. Easy induction. \square

THEOREM 3. Let P be a Flowchart program. Define relations B_i as in Definition 3 and relations C_i as in Figure 17. Then $B_i = C_i$ for all $i \geq 0$.

PROOF. By induction on i . The base case is trivial. For the inductive step, assume first $(b, b') \in B_{i+1}$. For $1 \leq j \leq |\text{succ}(b)|$ we have by definition $(\text{succ}_j(b), \text{succ}_j(b')) \in B_i$. We find inductively $(\text{succ}_j(b), \text{succ}_j(b')) \in C_i$, and it follows that $\alpha(\text{succ}_j(b)) = \alpha(\text{succ}_j(b'))$, hence $s(b) = s(b')$. By Lemma 2, $(b, b') \in B_i$, so we find inductively $(b, b') \in C_i$. Since both $(b, b') \in C_i$ and $s(b) = s(b')$, we must have $(b, b') \in C_{i+1}$ or C_{i+1} is not maximal.

Assume instead $(b, b') \in C_{i+1}$. By definition, $s(b) = s(b')$, so $\alpha(\text{succ}_j(b)) = \alpha(\text{succ}_j(b'))$ for $1 \leq j \leq |\text{succ}(b)|$, thus $(\text{succ}_j(b), \text{succ}_j(b')) \in C_i$. We find inductively that $(\text{succ}_j(b), \text{succ}_j(b')) \in B_i$ for $1 \leq j \leq |\text{succ}(b)|$. Noting $b \equiv b'$ since $(b, b') \in C_{i+1}$, we have $(b, b') \in B_{i+1}$ \square

Termination of the algorithm is obvious since the equivalence classes of the C_i must eventually become trivial. It is easy to see that the C_i and B_i arrive at a fixed point simultaneously. Thus, the following corollary holds:

COROLLARY 2. For any program P , the algorithm of Figure 17 computes the bisimulation equivalence on $P \times P$.