Proceedings of the MSPLS Spring ’98 Workshop

Saturday, 16 May 1998, Loyola University Chicago, Chicago, IL

Gerald Baumgartner(*) and Konstantin Läufer(**) (eds.)
Technical Report OSU-CISRC-6/98-TR21 (PDF Format)
Dept. of Computer and Information Sciences
The Ohio State University
URL: http://www.cis.ohio-state.edu/~gb/MSPLS/Spring98/

26 June 1998

Foreword

Historically, the aim of MSPLS Workshops has been to allow an informal exchange of ideas in all areas of programming languages and systems among researchers and practitioners from midwestern universities and companies. Contributions span the range from presentations on preliminary research results to introductory lectures on ongoing research efforts.

This report is a collection of abstracts, extended abstracts, and/or papers of the talks given at the Spring '98 Workshop of the Midwest Society for Programming Languages and Systems, which took place at Loyola University Chicago, Water Tower Campus, on Saturday, 16 May 1998, following the IEEE Computer Society 1998 International Conference on Computer Languages (ICCL '98).

Program

Attendees

The list of attendees is only available to workshop attendees.

(*) Dept. of Computer and Information Science, The Ohio State University, 395 Dreese Lab., 2015 Neil Ave., Columbus, OH 43210-1277. Email: gb@cis.ohio-state.edu.

(**) Dept. of Mathematical and Computer Sciences, Loyola University of Chicago, 6525 North Sheridan Road, Chicago, IL 60626. Email: laufer@cs.luc.edu.

Gerald Baumgartner

MSPLS Spring ’98 Workshop Program

Saturday, 16 May 1998

1:30-2:00 Registration

2:00-3:30 Technical Session I

Building a Bridge between Pointer Aliases and Program Dependences
John L. Ross (Dept. of Computer Science, University of Chicago) and Mooly Sagiv, (Tel-Aviv University)
Abstract, Paper

Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis
Rebecca Hasti and Susan Horwitz (Computer Sciences Dept., University of Wisconsin - Madison)
Abstract, Paper

Edge Profiling vs. Path Profiling: The Showdown
Thomas Ball, Peter Mataga (Bell Laboratories, Lucent Technologies), and Mooly Sagiv (Tel-Aviv University)
Abstract, Paper

3:30-4:00 Break

4:00-5:30 Technical Session II

Action Transformation: An Application of Sort Inference
Kent Lee (Dept. of Computer Science, Luther College)
Abstract, Slides, Slide supplement: test0.action, Slide supplement: test0.sorts, Paper

Integration of Software Development Documents with the Software Concordance
Ethan Munson (Dept. of Electrical Engineering and Computer Science, University of Wisconsin - Milwaukee)
Abstract, Paper

Issues in the Design of SWAR Programming Models
Randall J. Fisher and Hank Dietz (Dept. of Electrical and Computer Engineering, Purdue University)
Abstract

5:30-6:00

Business Meeting

Election of Gerald Baumgartner and Konstantin Läufer as new MSPLS co-presidents.

Selection of De Paul University, Chicago, as the meeting site for the Fall '98 Workshop. The workshop will be held on Saturday, 10 October 1998. It will be hosted by Karen Bernstein (kbernstein@cs.depaul.edu).

6:00-9:00 Dinner

Gerald Baumgartner

MSPLS Spring '98 Workshop Abstract

Building a Bridge between Pointer Aliases and Program Dependences

John L. Ross (Dept. of Computer Science, University of Chicago) and Mooly Sagiv, (Tel-Aviv University)

In this paper we present a surprisingly simple reduction of the program dependence problem to the may-alias problem. While both problems are undecidable, providing a bridge between them has great practical importance. Program dependence information is used extensively in compiler optimizations, automatic program parallelizations, code scheduling in super-scalar machines, and in software engineering tools such as code slicers. When working with languages that support pointers and references, these systems are forced to make very conservative assumptions. This leads to many superfluous program dependences and limits compiler performance and the usability of software engineering tools. Fortunately, there are many algorithms for computing conservative approximations to the may-alias problem. The reduction has the important property of always computing conservative program dependences when used with a conservative may-alias algorithm. We believe that the simplicity of the reduction and the fact that it takes linear time may make it practical for realistic applications.

Gerald Baumgartner

MSPLS Spring ’98 Workshop Abstract

Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis

Rebecca Hasti and Susan Horwitz (Computer Sciences Dept., University of Wisconsin - Madison)

A pointer-analysis algorithm can be either flow-sensitive or flow-insensitive. While flow-sensitive analysis usually provides more precise information, it is also usually considerably more costly in terms of time and space. This talk will present another option in the form of an algorithm that can be ‘tuned’ to provide a range of results that fall between the results of flow-insensitive and flow-sensitive analysis. The algorithm combines a flow-insensitive pointer analysis with static single assignment (SSA) form and uses an iterative process to obtain progressively better results.

Gerald Baumgartner

MSPLS Spring ’98 Workshop Abstract

Edge Profiling vs. Path Profiling: The Showdown

Thomas Ball, Peter Mataga (Bell Laboratories, Lucent Technologies), and Mooly Sagiv (Tel-Aviv University)

Edge profiles are the traditional control flow profile of choice for profile-directed compilation. They have been the basis of path-based optimizations that select "hot" paths, even though edge profiles contain strictly less information than path profiles. Recent work on path profiling has suggested that path profiles are superior to edge profiles in practice.

We present theoretic and algorithmic results that may be used to determine when an edge profile is a good predictor of hot paths (and what those hot paths are) and when it is a poor predictor. Our algorithms efficiently compute sets of *definitely* and *potentially* hot paths in a graph annotated with an edge profile. A definitely hot path has a frequency greater than some non-zero lower bound in all path profiles that induce a given edge profile.

Experiments on the SPEC95 benchmarks show that a huge percentage of the execution frequency in these programs is dominated by definitely hot paths (on average, 84% for FORTRAN benchmarks and 76% for C benchmarks). We also show that various hot path selection algorithms based on edge profiles work extremely well in most cases, but that path profiling is needed in some cases. These results indicate the usefulness of our algorithms for characterizing edge profiles and selecting hot paths.

Gerald Baumgartner

MSPLS Spring ’98 Workshop Abstract

Action Transformation: An Application of Sort Inference

Kent Lee (Dept. of Computer Science, Luther College)

Action Semantics is a formal method of defining programming language semantics in which actions describe the manipulation of three entities: transients, bindings, and storage. An Action Semantic Description for a programming language translates a program in the source language to a corresponding action. However, due to the high-level nature of Action Semantics, actions cannot generally be directly translated into efficient code in an Action Semantics-based compiler. However, by applying sort inference to an action, it is possible to transform it into an action that can be translated into efficient code. This talk will demonstrate the problem and solution by giving an example of a simple program and it’s compilation to efficient code using an Action Semantics-based compiler.

Gerald Baumgartner

MSPLS Spring ’98 Workshop Abstract

Integration of Software Development Documents with the Software Concordance

Ethan Munson (Dept. of Electrical Engineering and Computer Science, University of Wisconsin - Milwaukee)

Large software projects produce many documents of many different types that, together, describe the plans for, implementation of, and experience with the program. Ensuring that these documents do not have serious conflicts with each other is a critical task in any real software system, but requires a level of integration among the different types of software documents that is not possible with current development tools.

Our research project is now beginning work to identify the document representations, software tools, and user interface services that are required to integrate program source code and all other software documents into a seamless, interconnected whole. The project is producing a prototype development environment, called the Software Concordance, that allows arbitrary multimedia documentation and hyperlinks inside program source files and additional tools for managing and understanding the relationships between software documents. Successful creation of the Software Concordance will require solutions to problems of document representation, integration of incremental parsing and formatting services, fine-grained version control, and tools for analysis of networks of time-stamped, directed hypertext links. It is our hope that this research will break down the barriers that currently exist between program source code and the natural language documents that motivate, evaluate, and explain it.

In this talk, I will present our vision for the Software Concordance and describe the research problems that we will be investigating.

Gerald Baumgartner

MSPLS Spring '98 Workshop Abstract

Issues in the Design of SWAR Programming Models

Randall J. Fisher and Hank Dietz (Dept. of Electrical and Computer Engineering, Purdue University)

Over the past few years, high-end microprocessor designs have added support for a version of SIMD (Single Instruction, Multiple Data) parallel execution that can improve specific multimedia operations, yet can be implemented without completely restructuring the microprocessor design. This version of SIMD, which we call SWAR (SIMD Within A Register), uses most of the existing datapaths of the microprocessor, but allows registers and datapaths to be logically partitioned into fields on which operations SIMD parallel operations can be performed.

AMD/Cyrix/Intel MMX, Sun SPARC V9 VIS, HP PA-RISC MAX, DEC Alpha MAX, SGI MIPS MDMX, and now Motorola PowerPC AltiVec, are all SWAR extensions, but they are all different and somewhat quirky, initially intended only to be used by hand-writing assembly-level code. It is also possible to obtain speedup using software SWAR techniques with ordinary processors. This talk discusses the design of a portable high-level programming model for SWAR and the techniques needed to compile programs written in such a language into efficient implementations on any of these types of processors.

Gerald Baumgartner

Building a Bridge between Pointer Aliases and Program Dependences ⋆

John L. Ross[1] and Mooly Sagiv[2]

[1] University of Chicago, e-mail: johnross@cs.uchicago.edu
[2] Tel-Aviv University, e-mail: sagiv@math.tau.ac.il

Abstract. In this paper we present a surprisingly simple reduction of the program dependence problem to the may-alias problem. While both problems are undecidable, providing a bridge between them has great practical importance. Program dependence information is used extensively in compiler optimizations, automatic program parallelizations, code scheduling in super-scalar machines, and in software engineering tools such as code slicers. When working with languages that support pointers and references, these systems are forced to make very conservative assumptions. This leads to many superfluous program dependences and limits compiler performance and the usability of software engineering tools. Fortunately, there are many algorithms for computing conservative approximations to the may-alias problem. The reduction has the important property of always computing conservative program dependences when used with a conservative may-alias algorithm. We believe that the simplicity of the reduction and the fact that it takes linear time may make it practical for realistic applications.

1 Introduction

It is well known that programs with pointers are hard to understand, debug, and optimize. In recent years many interesting algorithms that conservatively analyze programs with pointers have been published. Roughly speaking, these algorithms [19, 20, 25, 5, 16, 17, 23, 8, 6, 9, 14, 27, 13, 28] conservatively solve the may-alias problem, i.e., the algorithms are sometimes able to show that two pointer access paths never refer to the same memory location at a given program point.

However, may-alias information is insufficient for compiler optimizations, automatic code parallelizations, instruction scheduling for super-scalar machines, and software engineering tools such as code slicers. In these systems, information about the program dependences between *different* program points is required. Such dependences can be uniformly modeled by the program dependence graph (see [21, 26, 12]).

In this paper we propose a simple yet powerful approach for finding program dependences for programs with pointers:

⋆ Partially supported by Binational Science Foundation grant No. 9600337

Given a program $\mathcal{P}$, we generate a program $\mathcal{P}'$ (hereafter also referred to as the *instrumented* version of $\mathcal{P}$) which simulates $\mathcal{P}$. The program dependences of $\mathcal{P}$ can be computed by applying an arbitrary conservative may-alias algorithm to $\mathcal{P}'$.

We are reducing the program dependence problem, a problem of great practical importance, to the may-alias problem, a problem with many competing solutions. The reduction has the property that as long as the may-alias algorithm is conservative, the dependences computed are also conservative. Furthermore, there is no loss of precision beyond that introduced by the chosen may-alias algorithm. Since the reduction is quite efficient (linear in the program size), it should be possible to integrate our method into compilers, program slicers, and other software tools.

1.1 Main Results and Related Work

The major results in this paper are:

- The unification of the concepts of program dependences and may-aliases. While these concepts are seemingly different, we provide linear reductions between them. Thus may-aliases can be used to find program dependences and program dependences can be used to find may-aliases.
- A solution to the previously open question about the ability to use "store-less" (see [8–10]) may-alias algorithms such as [9, 27] to find dependences.
  One of the simplest store-less may alias algorithm is due to Gao and Hendren [14]. In [15], the algorithm was generalized to compute dependences by introducing new names. Our solution implies that there is no need to re-develop a new algorithm for every may-alias algorithm. Furthermore, we believe that our reduction is actually simpler to understand than the names introduced in [15] since we are proving program properties instead of modifying a particular approximation algorithm.
- Our limited experience with the reduction that indicates that store-less may-alias algorithms such as [9, 27] yield quite precise dependence information.
- The provision of a method to compare the time and precision of different may-alias algorithms by measuring the number of program dependences reported. This metric is far more interesting than just comparing the number of may-aliases as done in [23, 11, 31, 30, 29].

Our program instrumentation closely resembles the "instrumented semantics" of Horwitz, Pfeiffer, and Reps [18]. They propose to change the program semantics so that the interpreter will carry-around program statements. We instrument the program itself to locally record statement information. Thus, an arbitrary may-alias algorithm can be used on the instrumented program without modification. In contrast, Horwitz, Pfeiffer, and Reps proposed modifications to the specific store based may-alias algorithm of Jones and Muchnick [19] (which is imprecise and doubly exponential in space).

An additional benefit of our shift from semantics instrumentation into a program transformation is that it is easier to understand and to prove correct. For example, Horwitz, Pfeiffer, and Reps, need to show the equivalence between the original and the instrumented program semantics and the instrumentation properties. In contrast, we show that the instrumented program simulates the original program and the properties of the instrumentation.

Finally, program dependences can also be conservatively computed by combining side-effect analysis [4, 7, 22, 6] with reaching definitions [2] or by combining conflict analysis [25] with reaching definitions as done in [24]. However, these techniques are extremely imprecise when recursive data structures are manipulated. The main reason is that it is hard to distinguish between occurrences of the same heap allocated run-time location (see [6, Section 6.2] for an interesting discussion).

1.2 Outline of the rest of this paper

In Section 2.1, we describe the simple Lisp-like language that is used throughout this paper. The main features of this language are its dynamic memory, pointers, and destructive assignment. The use of a Lisp-like language, as opposed to C, simplifies the presentation by avoiding types and the need to handle some of the difficult aspects of C, such as pointer arithmetic and casting.

In Section 2.2, we recall the definition of flow dependences. In Section 2.3 the may-alias problem is defined.

In Section 3 we define the instrumentation. We show that the instrumented program simulates the execution of the original program. We also show that for every run-time location of the original program the instrumented program maintains the history of the statements that last wrote into that location. These two properties allow us to prove that may-aliases in the instrumented program precisely determine the flow dependences in the original program.

In Section 4, we discuss the program dependences computed by some may-alias algorithms on instrumented programs. Finally, Section 5, contains some concluding remarks.

2 Preliminaries

2.1 Programs

Our illustrative language (following [19, 5]) combines an Algol-like language for control flow and functions, Lisp-like memory access, and explicit destructive assignment statements. The atomic statements of this language are shown in Table 1. Memory access paths are represented by $\langle Access \rangle$. Valid expressions are represented by $\langle Exp \rangle$. We allow arbitrary control flow statements using conditions $\langle Cond \rangle$[1]. Additionally all statements are labeled.

[1] Arbitrary expressions and procedures can be allowed as long as it is not possible to observe actual run-time locations.

Figure 1 shows a program in our language that is used throughout this paper as a running example. This program reads atoms and builds them into a list by destructively updating the *cdr* of the tail of the list.

```
      program DestructiveAppend()
      begin
$s_1$:      new( head )
$s_2$:      read( head.car )
$s_3$:      head.cdr := nil
$s_4$:      tail := head
$s_5$:      while( tail.car ≠ 'x' ) do
$s_6$:           new( temp )
$s_7$:           read( temp.car )
$s_8$:           temp.cdr := nil
$s_9$:           tail.cdr := temp
$s_{10}$:          tail := tail.cdr
           od
$s_{11}$:     write( head.car )
$s_{12}$:     write( tail.car )
      end.
```

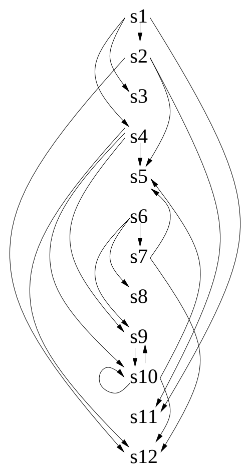


Fig. 1. A program that builds a list by destructively appending elements to *tail* and its flow dependences.

2.2 The Program Dependence Problem

Program dependences can be grouped into flow dependences (def-use), output dependences (def-def), and anti-dependences (use-def) [21, 12]. In this paper, we focus on flow dependences between program statements. The other dependences are easily handled with only minor modifications to our method.

Table 1. An illustrative language with dynamic memory and destructive updates.

$\langle St\rangle ::= s_i\!:\langle Access\rangle := \langle Exp\rangle$
$\langle St\rangle ::= s_i\!:\mathbf{new}(\langle Access\rangle)$
$\langle St\rangle ::= s_i\!:\mathbf{read}(\langle Access\rangle)$
$\langle St\rangle ::= s_i\!:\mathbf{write}(\langle Exp\rangle)$
$\langle Access\rangle ::= \mathit{variable} \mid \langle Access\rangle.\langle Sel\rangle$
$\langle Exp\rangle ::= \langle Access\rangle \mid \mathit{atom} \mid \mathbf{nil}$
$\langle Sel\rangle ::= \mathit{car} \mid \mathit{cdr}$
$\langle Cond\rangle ::= \langle Exp\rangle = \langle Exp\rangle$
$\langle Cond\rangle ::= \langle Exp\rangle \neq \langle Exp\rangle$

Our language allows programs to explicitly modify their store through pointers. Because of this we must phrase the definition of flow dependence in terms of memory *locations* (cons-cells) and not variable names. We shall borrow the following definition for flow dependence:

Definition 1 ([18]). *Program point q has a flow dependence on program point p if p writes into memory location loc that q reads, and there is no intervening write into loc along an execution path by which q is reached from p.*

Figure 1 also shows the flow dependences for the running example program. Notice that s_{11} is flow dependent on only s_1 and s_2, while s_{12} is flow dependent on s_2, s_4, s_7, and s_{10}. This information could be used by slicing tools to find that the loop need not be executed to print *head.car* in s_{11}, or by an instruction scheduler to reschedule s_{11} for anytime after s_2. Also, s_3 and s_8 have no statements dependent on them, making them candidates for elimination. Thus, even in this simple example, knowing the flow dependences would potentially allow several code transformations.

Since determining the exact flow dependences in an arbitrary program is undecidable, approximation algorithms must be used. A flow dependence approximation algorithm is *conservative* if it always finds a superset of the true flow dependences.

2.3 The May-Alias Problem

The may-alias problem is to determine whether two access-paths, at a given program point, could denote the same cons-cell.

Definition 2. *Two access-paths are may-aliases at program point p, if there exists an execution path to program point p where both denote the same cons-cell.*

In the running example program, *head.cdr.cdr* and *tail* are may-aliases at s_6 since before the third iteration these access paths denote the same cons-cell. However, *tail.cdr.cdr* is not a may-alias to *head* since they can never denote the same cons-cell.

Since the may-alias problem is undecidable, approximation algorithms must be used. A may-alias approximation algorithm is *conservative* if it always finds a superset of the true may-aliases.

3 The Instrumentation

In this section the instrumentation is defined. For notational simplicity, $\mathcal{P}$ stands for an arbitrary fixed program, and $\mathcal{P}'$ stands for its instrumented version.

3.1 The Main Idea

The program $\mathcal{P}'$ simulates all the execution sequences of $\mathcal{P}$. Additionally, the "observable" properties of $\mathcal{P}$ are preserved.

Most importantly, $\mathcal{P}'$ records for every variable v, the statement from $\mathcal{P}$ that last wrote into v. This "instrumentation information" is recorded in $v.car$ (while storing the original content of v in $v.cdr$). This "totally static" instrumentation[2] allows program dependences to be recovered by may-alias queries on $\mathcal{P}'$.

More specifically, for every statement in $\mathcal{P}$ there is an associated cons-cell in $\mathcal{P}'$. We refer to these as *statement* cons-cells. Whenever a statement s_i assigns a value into a variable v, $\mathcal{P}'$ allocates a cons-cell that we refer to as an *instrumentation* cons-cell. The *car* of this instrumentation cons-cell always points to the statement cons-cell associated with s_i. Thus there is a flow dependence from a statement p to q: $x := y$ in $\mathcal{P}$ if and only if $y.car$ can point to the statement cons-cell associated with p in $\mathcal{P}'$. Finally, we refer to the *cdr* of the instrumentation cons-cell as the *data* cons-cell. The data cons-cell is inductively defined:

- If s_i is an assignment $s_i\colon v := A$ for an atom, A, then the data cell is A.
- If s_i is an assignment $s_i\colon v := v'$ then the data cell denotes $v'.cdr$.
- If the statement is $s_i\colon new(v)$, then the data cons-cell denotes a newly allocated cons-cell. Thus $\mathcal{P}'$ allocates two cons-cells for this statement.

In general, there is an inductive syntax directed definition of the data cells formally defined by the function *txe* defined in Table 3.

3.2 The Instrumentation of the Running Example

To make this discussion concrete, Figure 2 shows the beginning of the running example program and its instrumented version. Figure 3 shows the stores of both the programs just before the loop (on the input beginning with 'A'). The cons-cells in this figure are labeled for readability only.

The instrumented program begins by allocating one statement cons-cell for each statement in the original program. Then, for every statement in the original program, the instrumented statement block in the instrumented program records the last wrote-statement and the data. The variable *_rhs* is used as a temporary to store the right-hand side of an assignment to allow the same variable to be used on both sides of an assignment.

Let us now illustrate this for the statements s_1 through s_4 in Figure 3.

- In the original program, after s_1, *head* points a new uninitialized cons-cell, c_1. In the instrumented program, after the block of statements labeled by s_1, *head* points to an instrumentation cons-cell, i_1, *head.car* points to the statement cell for s_1, and *head.cdr* points to c_1'.

[2] In contrast to dynamic program slicing algorithms that record similar information using hash functions, e.g., [1].

- In the original program, after s_2, *head.car* points to the atom A. In the instrumented program, after the block of statements labeled by s_2, *head.cdr.car* points to an instrumentation cons-cell, i_2, *head.cdr.car.car* points to the statement cell for s_2, and *head.cdr.car.cdr* points to A.
- In the original program, after s_3, *head.cdr* points to nil. In the instrumented program, after the block of statements labeled by s_3, *head.cdr.cdr* points to an instrumentation cons-cell, i_3, *head.cdr.cdr.car* points to the statement cell for s_3, and *head.cdr.cdr.cdr* points to nil.
- In the original program, after s_4, *tail* points to the cons-cell c_1. In the instrumented program, after the block of statements labeled by s_4, *tail* points to an instrumentation cons-cell, i_4, *tail.car* points to the statement cell for s_4, and *tail.cdr* points to c_1'. Notice how the sharing of the r-values of *head* and *tail* is preserved by the transformation.

3.3 A Formal Definition of the Instrumentation

Formally, we define the instrumentation as follows:

Definition 3. *Let $\mathcal{P}$ be a program in the form defined in Table 1. Let $s_1, s_2, \ldots, s_n$ be the statement labels in $\mathcal{P}$. The instrumented program $\mathcal{P}'$ is obtained from $\mathcal{P}$ starting with a prolog of the form* $\mathbf{new}(ps_i)$ *where $i = 1, 2 \ldots n$. After the prolog, we rewrite $\mathcal{P}$ according to the translation rules shown in Table 2 and Table 3.*

Example 4. In the running example program (Figure 2), in $\mathcal{P}$, s_{11} writes *head.car* and in $\mathcal{P}'$, s_{11} writes *head.cdr.car.cdr*. This follows from:
$txe(head.car) = txa(head.car).cdr = txa(head).cdr.car.cdr = head.cdr.car.cdr$

3.4 Properties of the Instrumentation

In this section we show that the instrumentation has reduced the flow dependence problem to the may-alias problem. First the simulation of $\mathcal{P}$ by $\mathcal{P}'$ is shown in the Simulation Theorem. This implies that the instrumentation does not introduce any imprecision into the flow dependence analysis. We also show the Last Wrote Lemma which states that the instrumentation maintains the needed invariants. Because of the Simulation Theorem, and the Last Wrote Lemma, we are able to conclude that:

1. exactly all the flow dependences in $\mathcal{P}$ are found using a may-alias oracle on $\mathcal{P}'$.
2. using any conservative may-alias algorithm on $\mathcal{P}'$ always results in conservative flow dependences for $\mathcal{P}$.

Intuitively, by simulation, we mean that at every label of $\mathcal{P}$ and $\mathcal{P}'$, all the "observable properties" are preserved in $\mathcal{P}'$, given the same input. In our case, observable properties are:

- r-values printed by the write statements

	program *DestructiveAppend()*		**program** *InstrumentedDestructiveAppend()*
	begin		**begin**
			new(ps_i) $\forall i \in \{1, 2, \ldots, 12\}$
s_1:	**new**(*head*)	s_1:	**new**(*head*)
			head.car := *ps1*
			new(*head.cdr*)
s_2:	**read**(*head.car*)	s_2:	**new**(*head.cdr.car*)
			head.cdr.car.car := *ps2*
			read(*head.cdr.car.cdr)*
s_3:	*head.cdr* := **nil**	s_3:	*_rhs* := **nil**
			new(*head.cdr.cdr*)
			head.cdr.cdr.car := *ps3*
			head.cdr.cdr.cdr := *_rhs*
s_4:	*tail* := *head*	s_4:	*_rhs* := *head.cdr*
			new(*tail*)
			tail.car := *ps4*
			tail.cdr := *_rhs*
s_5:	**while**(*tail.car* $\neq$ 'x') **do**	s_5:	**while**(*tail.cdr.car.cdr* $\neq$ 'x') **do**

Fig. 2. The beginning of the example program and its corresponding instrumented program.

Table 2. The translation rules that define the instrumentation excluding the prolog. For simplicity, every assignment allocates a new instrumentation cons-cell. The variable *_rhs* is used as a temporary to store the right-hand side of an assignment to allow the same variable to be used on both sides of an assignment.

s_i: $\langle Access\rangle$:=$\langle Exp\rangle$	$\Longrightarrow$	s_i: *_rhs* :=$txe(\langle Exp\rangle)$
		new$(txa(\langle Access\rangle))$
		$txa(\langle Access\rangle).car$:=$ps_i$
		$txa(\langle Access\rangle).cdr$:=*_rhs*
s_i: **new**$(\langle Access\rangle)$	$\Longrightarrow$	s_i: **new**$(txa(\langle Access\rangle))$
		$txa(\langle Access\rangle).car$:=$ps_i$
		new$(txa(\langle Access\rangle).cdr)$
s_i: **read**$(\langle Access\rangle)$	$\Longrightarrow$	s_i: **new**$(txa(\langle Access\rangle))$
		$txa(\langle Access\rangle).car$:=$ps_i$
		read$(txa(\langle Access\rangle).cdr)$
s_i: **write**$(\langle Exp\rangle)$	$\Longrightarrow$	s_i: **write**$(txa(\langle Exp\rangle))$
$\langle Exp_1\rangle = \langle Exp_2\rangle$	$\Longrightarrow$	$txe(\langle Exp_1\rangle) = txe(\langle Exp_2\rangle)$
$\langle Exp_1\rangle \neq \langle Exp_2\rangle$	$\Longrightarrow$	$txe(\langle Exp_1\rangle) \neq txe(\langle Exp_2\rangle)$

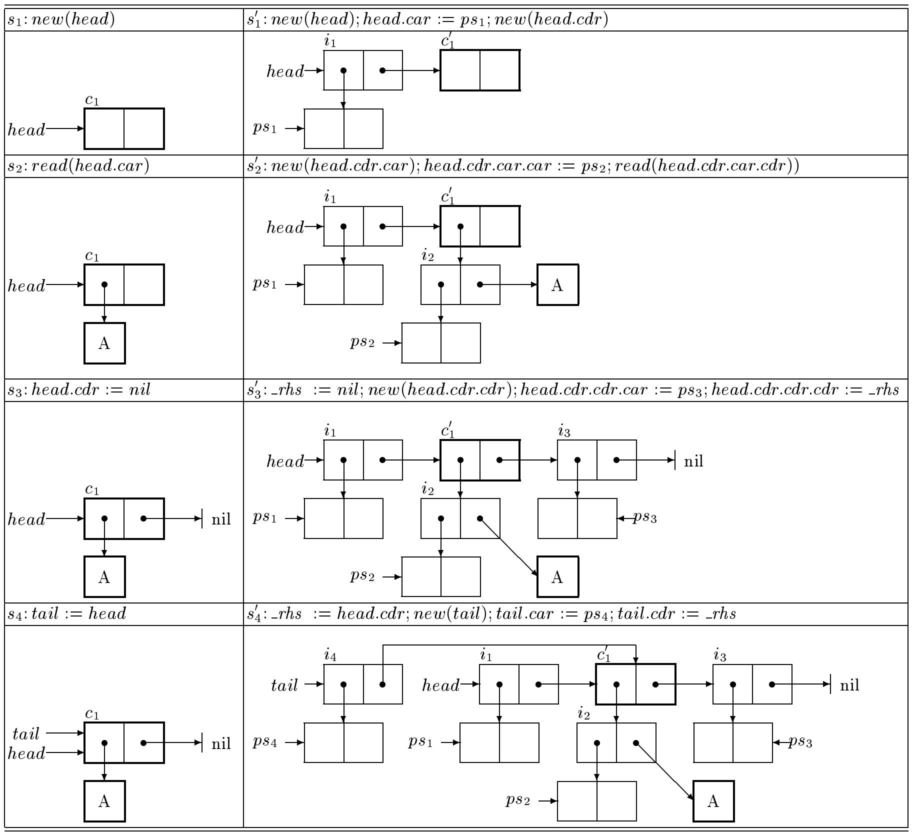

s_1: $new(head)$
s'_1: $new(head); head.car := ps_1; new(head.cdr)$
s_2: $read(head.car)$
s'_2: $new(head.cdr.car); head.cdr.car.car := ps_2; read(head.cdr.car.cdr))$
s_3: $head.cdr := nil$
s'_3: $\_rhs := nil; new(head.cdr.cdr); head.cdr.cdr.car := ps_3; head.cdr.cdr.cdr := \_rhs$
s_4: $tail := head$
s'_4: $\_rhs := head.cdr; new(tail); tail.car := ps_4; tail.cdr := \_rhs$
$head$
$tail$
c_1
c'_1
i_1
i_2
i_3
i_4
ps_1
ps_2
ps_3
ps_4
A
nil

Fig. 3. The store of the original and the instrumented running example programs just before the loop on an input list starting with 'A'. For visual clarity, statement cons-cells not pointed to by an instrumentation cons-cell are not shown. Also, cons-cells are labeled and highlighted to show the correspondence between the stores of the original and instrumented programs.

Table 3. The function *txa* which maps an access path in the original program into the corresponding access path in the instrumented program. The function *txe* maps an expression into the corresponding expression in the instrumented program.

$txa(variable)$	$= variable$
$txa(\langle Access\rangle.\langle Sel\rangle)$	$= txa(\langle Access\rangle).cdr.\langle Sel\rangle$
$txe(\langle Access\rangle)$	$= txa(\langle Access\rangle).cdr$
$txe(atom)$	$= atom$
$txe(nil)$	$= nil$

– equalities of r-values

In particular, the execution sequences of $\mathcal{P}$ and $\mathcal{P}'$ at every label are the same. This discussion motivates the following definition:

Definition 5. *Let S be an arbitrary sequence of statement labels in $\mathcal{P}$ e_1, e_2 be expressions, and I be an input vector. We denote by $I, S \overset{\mathcal{P}}{\models} e_1 = e_2$ the fact that the input I causes S to be executed in $\mathcal{P}$, and in the store after S, the r-values of e_1 and e_2 are equal.*

Example 6. In the running example, *head.cdr.cdr* and *tail* denote the same cons-cell before the third iteration for inputs lengths of four or more. Therefore, $I, [s_1, s_2, s_3, s_4, s_5]([s_6, s_7, s_8, s_9, s_{10}])^2 \overset{\mathcal{P}}{\models} \mathit{head.cdr.cdr} = \mathit{tail}$

Theorem 7. (Simulation Theorem) *Given input I, expressions e_1 and e_2, and sequence of statement labels S:*

$$I, S \overset{\mathcal{P}}{\models} e_1 = e_2 \iff I, S \overset{\mathcal{P}'}{\models} \mathit{txe}(e_1) = \mathit{txe}(e_2)$$

Example 8. In the running example, before the first iteration, in the last box of Figure 3, *head* and *tail* denote the same cons-cell and *head.cdr* and *tail.cdr* denote the same cons-cell. Also, in the instrumented program, *head.cdr.cdr.cdr.cdr* and *tail.cdr* denote the same cons-cell before the third iteration for inputs of length four or more. Therefore, as expected from Example 6,
$I, [s_1, s_2, s_3, s_4, s_5]([s_6, s_7, s_8, s_9, s_{10}])^2 \overset{\mathcal{P}'}{\models} \mathit{txe}(\mathit{head.cdr.cdr}) = \mathit{txe}(\mathit{tail})$

The utility of the instrumentation is captured in the following lemma.

Lemma 9. (Last Wrote Lemma) *Given input I, sequence of statement labels S, and an access path a, the input I leads to the execution of S in $\mathcal{P}$ in which the last statement that wrote into a is s_i if and only if*

$$I, S \overset{\mathcal{P}'}{\models} \mathit{txa}(a).\mathit{car} = \mathit{ps}_i.$$

Example 10. In the running example, before the first iteration, in the last box of Figure 3, we have $I, [s_1, s_2, s_3, s_4, s_5] \overset{\mathcal{P}'}{\models} \mathit{txa}(\mathit{head}).\mathit{car} = \mathit{ps}_1$ since s_1 is the statement that last wrote into *head*. Also, for input list $I = ['A', 'x']$, we have: $I, [s_1, s_2, s_3, s_4, s_5, s_{11}, s_{12}] \overset{\mathcal{P}'}{\models} \mathit{txa}(\mathit{tail.car}).\mathit{car} = \mathit{ps}_2$ since for such input s_2 last wrote into *tail.car* (through the assignment to *head.car*).

A single statement in our language can read from many memory locations. For example, in the running example program, statement s_5 reads from *tail* and *tail.car*. The complete read-sets for the statements in our language are shown in Tables 4 and 5.

We are now able to state the main result.

Table 4. Read-sets for the statements in our language.

s_i: $\langle Access\rangle := \langle Exp\rangle$	$rsa(\langle Access\rangle) \cup rse(\langle Exp\rangle)$
s_i: **new**$(\langle Access\rangle)$	$rsa(\langle Access\rangle)$
s_i: **read**$(\langle Access\rangle)$	$rse(\langle Access\rangle)$
s_i: **write**$(\langle Exp\rangle)$	$rse(\langle Exp\rangle)$
$\langle Exp_1\rangle = \langle Exp_2\rangle$	$rse(\langle Exp_1\rangle) \cup rse(\langle Exp_2\rangle)$
$\langle Exp_1\rangle \neq \langle Exp_2\rangle$	$rse(\langle Exp_1\rangle) \cup rse(\langle Exp_2\rangle)$

Table 5. An inductive definition of rsa, the read-set for access-paths, and rse, the read-set for expressions.

$rsa(variable)$	$= \emptyset$
$rsa(\langle Access\rangle.\langle Sel\rangle)$	$= rsa(\langle Access\rangle) \cup \{\langle Access\rangle\}$
$rse(variable)$	$= \{variable\}$
$rse(\langle Access\rangle.\langle Sel\rangle)$	$= rse(\langle Access\rangle) \cup \{\langle Access\rangle.\langle Sel\rangle\}$
$rse(atom)$	$= \emptyset$
$rse(nil)$	$= \emptyset$

Theorem 11. (Flow Dependence Reduction) *Given program $\mathcal{P}$, its instrumented version $\mathcal{P}'$, and any two statement labels p and q. There is a flow dependence from p to q (in $\mathcal{P}$) if and only if there exists an access path, a, in the read-set of q, s.t. ps_p is a may-alias of $txa(a).car$ at q in $\mathcal{P}'$.*

Example 12. To find the flow dependences of s_5 in the running example:
$s_5 : while(tail.car \neq `x')$
First Tables 4 and 5 are used to determine the read-set of s_5:

$$\begin{aligned} rse(\langle Exp_1\rangle) \cup rse(\langle Exp_2\rangle) &= rse(tail.car) \cup rse(`x') = rse(tail) \cup \{tail.car\} \cup \emptyset \\ &= \{tail\} \cup \{tail.car\} = \{tail, tail.car\}. \end{aligned}$$

Then $txa(a).car$ is calculated for each a in the read-set:
$txa(tail).car = tail.car$
$txa(tail.car).car = txa(tail).cdr.car.car = tail.cdr.car.car$
Next the may-aliases to $tail.car$ and $tail.cdr.car.car$ are calculated by any may-aliases algorithm. Finally s_5 is flow dependent on the statements associated with the statement cons-cells that are among the may-aliases found to $tail.car$ and $tail.cdr.car.car$.

The Read-Sets and May-Aliases for the running example are summarized in Table 6.

From a complexity viewpoint our method can be very inexpensive. The program transformation time and space are linear in the size of the original program. In applying Theorem 11 the number of times the may-alias algorithm is invoked is also linear in the size of the original program, or more specifically, proportional

Table 6. Flow dependence analysis of the running example using a may-alias oracle.

Stmt	**Read-Set**	**May-Aliases**
s_1	$\emptyset$	$\emptyset$
s_2	$\{head\}$	$\{(head.car, ps_1)\}$
s_3	$\{head\}$	$\{(head.car, ps_1)\}$
s_4	$\{head\}$	$\{(head.car, ps_1)\}$
s_5	$\{tail, tail.car\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}),$ $(tail.cdr.car.car, ps_2), (tail.cdr.car.car, ps_7)\}$
s_6	$\emptyset$	$\emptyset$
s_7	$\{temp\}$	$\{(temp.car, ps_6)\}$
s_8	$\{temp\}$	$\{(temp.car, ps_6)\}$
s_9	$\{tail, temp\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}), (temp.car, ps_6)\}$
s_{10}	$\{tail, tail.cdr\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}), (tail.cdr.cdr.car, ps_9)\}$
s_{11}	$\{head, head.car\}$	$\{(head.car, ps_1), (head.cdr.car.car, ps_2)\}$
s_{12}	$\{tail, tail.car\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}),$ $(tail.cdr.car.car, ps_2), (tail.cdr.car.car, ps_7)\}$

to the size of the read-sets. It is most likely that the complexity of the may-alias algorithm itself is the dominant cost.

4 Plug and Play

An important corollary of Theorem 11 is that an arbitrary conservative may-alias algorithm on $\mathcal{P}'$ yields a conservative solution to the flow dependence problem on $\mathcal{P}$. Since existing may-alias algorithms often yield results which are difficult to compare, it is instructive to consider the flow dependences computed by various algorithms on the running example program.

- The algorithm of [9] yields the may-aliases shown in column 3 of Table 6. Therefore, on this program, this algorithm yields the exact flow dependences shown in Figure 1.
- The more efficient may-alias algorithms of [23, 14, 30, 29] are useful to find flow dependences in programs with disjoint data structures. However, in programs with recursive data structures such as the running example, they normally yield many superfluous may-aliases leading to superfluous flow dependences. For example, [23] is not able to identify that *tail* points to an acyclic list. Therefore, it yields that *head.car* and ps_7 are may-aliases at s_{11}. Therefore, it will conclude that the value of *head.car* read at s_{11} may be written inside the loop (at statement s_7).
- The algorithm of [27] finds, in addition to the correct dependences, superfluous flow dependences in the running example. For example, it finds that s_5 has a flow dependence on s_8. This inaccuracy is attributable to the anonymous nature of the second cons-cell allocated with each new statement. There are two possible ways to remedy this inaccuracy:

- Modify the algorithm so that it is 2-bounded, i.e., also keeps track of *car* and *cdr* fields of variables. Indeed, this may be an adequate solution for general k-bounded approaches, e.g., [19] by increasing k to $2k$.
- Modify the transformation to assign unique names to these cons-cells. We have implemented this solution, and tested it using the PAG [3] implementation of the [27] algorithm[3] and found exactly all the flow dependences in the running example.

5 Conclusions

In this paper, we showed that may-alias algorithms can be used, without any modification, to compute program dependences. We are hoping that this will lead to more research in finding practical may-alias algorithms to compute good approximations for flow dependences.

For simplicity, we did not optimize the memory usage of the instrumented program. In particular, for every executed instance of a statement in the original program that writes to the store, the instrumented program creates a new instrumentation cons-cell. This extra memory usage is harmless to may-alias algorithms (for some algorithms it can even improve the accuracy of the analysis, e.g., [5]). In cases where the instrumented program is intended to be executed, it is possible to drastically reduce the memory usage through cons-cell reuse.

Finally, it is worthwhile to note that flow dependences can be also used to find may-aliases. Therefore, may-aliases are necessary in order to compute flow dependences. For example, Figure 4 contains a program fragment that provides the instrumentation to "check" if two program variables v_1 and v_2 are may aliases at program point p. This instrumentation preserves the meaning of the original program and has the property that v_1 and v_2 are may-aliases at p if and only if s_2 has a flow dependence on s_1.

```
p: if $v_1 \neq nil$ then
    $s_1$: $v_1.cdr := v_1.cdr$ fi
    if $v_2 \neq nil$ then
      $s_2$: $write(v_2.cdr)$ fi
```

Fig. 4. A program fragment such that v_1 and v_2 are may-aliases at p if and only if s_2 has a flow dependence on s_1.

Acknowledgments

We are grateful for the helpful comments of Thomas Ball, Michael Benedikt, Thomas Reps, and Reinhard Wilhelm for their comments that led to substantial

[3] On a SPARCstation 20, PAG used 0.53 seconds of cpu time.

improvements of this paper. We also would like to thank Martin Alt and Florian Martin for PAG, and their PAG implementation of [27] for a C subset.

References

1. H. Agrawal and J.R. Horgan. Dynamic program slicing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, volume 25 of *ACM SIGPLAN Notices*, pages 246–256, White Plains, New York, June 1990.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.
3. M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 33–50. Springer-Verlag, 1995.
4. J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *ACM Symposium on Principles of Programming Languages*, pages 29–41, New York, NY, 1979. ACM Press.
5. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 296–310, New York, NY, 1990. ACM Press.
6. J.D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, New York, NY, 1993. ACM Press.
7. K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 57–66, New York, NY, 1988. ACM Press.
8. A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, Washington, DC, 1992. IEEE Press.
9. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 230–241, New York, NY, 1994. ACM Press.
10. A. Deutsch. Semantic models and abstract interpretation for inductive data structures and pointers. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'95*, pages 226–228, New York, NY, June 1995. ACM Press.
11. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Languages Design and Implementation*, New York, NY, 1994. ACM Press.
12. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 3(9):319–349, 1987.
13. R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? In *ACM Symposium on Principles of Programming Languages*, New York, NY, January 1996. ACM Press.
14. R. Ghiya and L.J. Hendren. Connection analysis: A practical interprocedural heap analysis for c. In *Proc. of the 8th Intl. Work. on Languages and Compilers for Parallel Computing*, number 1033 in Lecture Notes in Computer Science, pages 515–534, Columbus, Ohio, August 1995. Springer-Verlag.

15. R. Ghiya and L.J. Hendren. Putting pointer analysis to work. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, January 1998.
16. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, Ithaca, N.Y., Jan 1990.
17. L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.
18. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Languages Design and Implementation*, volume 24 of *ACM SIGPLAN Notices*, pages 28–40, Portland, Oregon, June 1989. ACM Press.
19. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
20. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on Principles of Programming Languages*, pages 66–74, New York, NY, 1982. ACM Press.
21. D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages*, pages 207–218, New York, NY, 1981. ACM Press.
22. W. Land, B.G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 56–67, 1993.
23. W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, New York, NY, January 1991. ACM Press.
24. J.R. Larus. Refining and classifying data dependences. Unpublished extended abstract, Berkeley, CA, November 1988.
25. J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 21–34, New York, NY, 1988. ACM Press.
26. K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, New York, NY, 1984. ACM Press.
27. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *ACM Symposium on Principles of Programming Languages*, New York, NY, January 1996. ACM Press.
28. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 1997. To Appear.
29. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, 1997.
30. B. Steengaard. Points-to analysis in linear time. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, January 1996.
31. R.P. Willson and M.S. Lam. Efficient context-sensitive pointer analysis for c programs. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 1–12, La Jolla, CA, June 18-21 1995. ACM Press.

Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis [*]

Rebecca Hasti and Susan Horwitz
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
Electronic mail: {hasti, horwitz}@cs.wisc.edu

Abstract

A pointer-analysis algorithm can be either flow-sensitive or flow-insensitive. While flow-sensitive analysis usually provides more precise information, it is also usually considerably more costly in terms of time and space. The main contribution of this paper is the presentation of another option in the form of an algorithm that can be 'tuned' to provide a range of results that fall between the results of flow-insensitive and flow-sensitive analysis. The algorithm combines a flow-insensitive pointer analysis with static single assignment (SSA) form and uses an iterative process to obtain progressively better results.

1 Introduction

Having information about what pointer variables may point to is very useful (and often necessary) when performing many kinds of program analyses. Obviously, the better (or more precise) the information, the more useful the information is. A points-to analysis that takes into account the order in which statements may be executed (*i.e.,* a flow-sensitive analysis) generally provides more precise information than a flow-insensitive analysis; however, flow-sensitive analyses are considerably more costly in terms of time and/or space than flow-insensitive analyses. Thus, the options for pointer analysis one is generally presented with are: (1) flow-insensitive - faster but less precise; and (2) flow-sensitive - more precise but time/space consuming. The main contribution of this paper is the presentation of another option in the form of an algorithm that can be 'tuned' to provide a range of results. The algorithm combines a flow-insensitive pointer analysis with static single assignment (SSA) form and uses an iterative process to obtain progressively better results along the spectrum from flow-insensitive to flow-sensitive. The particular flow-insensitive analysis used will affect the precision of the final results. Whether it is possible to obtain results as precise as those obtained by a flow-sensitive analysis is an open question.

[*]This work was supported in part by the National Science Foundation under grant CCR-9625656, and by the Army Research Office under grant DAAH04-85-1-0482.

To appear in the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, June 1998, Montreal, Canada.

1.1 Flow-sensitive vs. flow-insensitive analysis

Program analyses may be categorized as either flow-sensitive or flow-insensitive. A flow-sensitive analysis takes into account the order in which the statements in the program may be executed; a flow-insensitive analysis does not. In other words, in a flow-sensitive analysis the program is handled as a *sequence* of statements while in a flow-insensitive analysis it is handled as a *set* of statements. Thus, a flow-sensitive analysis produces results at the statement level (*e.g.,* it may discover different properties of a variable p at each statement) whereas a flow-insensitive analysis produces results at the program level (*e.g.,* it can only discover properties of a variable p that hold for the entire program).

(Analyses can be further categorized as context-sensitive or context-insensitive. A context-sensitive analysis takes into account the fact that a function must return to the site of the most recent call; a context-insensitive analysis propagates information from a call site, through the called function, and back to *all* call sites. In this paper, all analyses are assumed to be context-insensitive.)

One way to think about flow-insensitive analysis is in terms of a variation on the standard dataflow framework [Kil73]. The standard framework includes:

1. a lattice of dataflow facts,
2. a set of monotonic dataflow functions,
3. a control flow graph (CFG),
4. a mapping that associates one dataflow function with each graph node (we use f_n to denote the function mapped to node n).

The ideal goal of a flow-sensitive analysis is to find the meet-over-all-paths solution to the dataflow problem [Kil73]. When this is not feasible (*e.g.,* when the functions are not distributive), an acceptable goal is to find the greatest solution (under the lattice ordering) to the following set of equations (one equation for each CFG node n):

$$n.fact = \sqcap_{m \in predecessors(n)} f_m(m.fact) \qquad (1)$$

(This equation is for a *forward* dataflow problem. The equation for a *backward* dataflow problem is similar, with successors used in place of predecessors.)

Flow-insensitive analysis uses the same framework, except that it uses a version of the CFG in which there is

$a = 1$	$a_0 = 1$	
$b = 2$	$b_0 = 2$	
$c = a + b$	$c_0 = a_0 + b_0$	
if (...)	if (...)	(1)
then $a = 3$	then $a_1 = 3$	
	$a_2 = \phi(a_1, a_0)$	
$d = b$	$d_0 = b_0$	
$c = a + b$	$c_1 = a_2 + b_0$	(2)
print(c)	print(c_1)	(3)
(a) Original	(b) SSA Form	

Figure 1: Example for constant propagation

an edge from each node to every other node (including itself) [Hor97]. Again, the ideal goal of the analysis is to find the meet-over-all-paths solution (in the modified CFG), and when this is not feasible, to find the greatest solution to the set of equations:

$$n.fact = \sqcap_{m \epsilon nodes(CFG)} f_m(m.fact) \qquad (2)$$

(Note that this framework is useful for *understanding* flow-insensitive analysis; actual algorithms do not involve creating this modified CFG or directly solving these equations.)

For the rest of this paper, when we refer to a flow-sensitive analysis, we mean an analysis that computes the greatest solution to the set of equations (1). Similarly, when we refer to a flow-insensitive analysis, we mean an analysis that computes the greatest solution to the set of equations (2).

Flow-sensitive analyses generally take more time and/or space than their flow-insensitive counterparts; however, the results are usually more precise. For example, consider constant propagation on the code fragment in Figure 1(a). A flow-sensitive constant-propagation analysis determines that:

- At[1] (1), $a = 1$, $b = 2$, $c = 3$
- At (2), $b = 2$, $c = 3$, $d = 2$
- At (3), $b = 2$, $d = 2$

and a flow-insensitive constant-propagation analysis determines that:

- $b = 2$, $d = 2$

Note that the results of the flow-insensitive analysis actually mean that at every point in the program, b is either uninitialized or has the value 2, and similarly for d. This kind of information is sufficient for most uses of the results of constant propagation (*e.g.,* replacing uses of constant variables with their values). Also note that, although the results of flow-insensitive analysis are not as precise as the results of flow-sensitive analysis, they do provide some useful information.

1.2 Using SSA form to improve flow-insensitive analysis

Static Single Assignment (SSA) form [CFR$^+$91] is a program representation in which variables are renamed (via subscripting) and new definitions inserted to ensure that:

1. Each variable x_i has exactly one definition site.
2. Each use of a variable x_i is reached by exactly one definition.

The new definitions (called ϕ nodes) are inserted in the CFG at those places reached by two (or more) definitions of a variable x (the join points) and are of the form:

$$x_k = \phi(x_{i_1}, x_{i_2}, \ldots, x_{i_n})$$

Figure 1(b) shows the SSA form for the example from Figure 1(a). Notice that once the program has been put into SSA form, flow-sensitive and flow-insensitive constant propagation identify the same instances of constant variables in the code. For example, both a flow-sensitive and a flow-insensitive analysis on Figure 1(b) produce the following results (the results are shown in flow-sensitive format; the results from flow-insensitive analysis hold not just at the point given, but at every point in the program):

- At (1), $a_0 = 1$, $b_0 = 2$, $c_0 = 3$
- At (2), $a_0 = 1$, $b_0 = 2$, $c_0 = 3$, $a_1 = 3$, $d_0 = 2$
- At (3), $a_0 = 1$, $b_0 = 2$, $c_0 = 3$, $a_1 = 3$, $d_0 = 2$

In other words, it does not matter whether the constant propagation analysis done on the SSA form is flow-sensitive or flow-insensitive. Thus, if the time and space required to translate a program into SSA form and then perform a flow-insensitive analysis are less than the time and space required to do a flow-sensitive analysis, this approach is a win.

1.3 Points-to analysis

The presence of pointers in a program makes it necessary to have information about what pointer variables may be pointing to in order to do many program analyses (such as constant propagation) correctly. Thus, a points-to analysis must first be done on a program before any further analyses are done. (There are two kinds of points-to analyses, may and must. Whenever we use 'points-to' we mean 'may-point-to'.) This points-to analysis may be flow-sensitive or flow-insensitive. For example, consider the code fragment in Figure 2(a).

A flow-sensitive points-to analysis determines the following points-to information at (*i.e.,* immediately before) each program point ($p \to a$ means that p might point to a, $p \to \{a, b\}$ means that p might point to a or to b):

Point	Points-to information
(2)	$a \to w$
(3)	$a \to w,\ p \to a$
(4)	$a \to x,\ p \to a$
(5)	$a \to x,\ p \to a,\ c \to x$
(6)	$a \to x,\ p \to a,\ c \to x$
(7)	$a \to \{x, y\},\ p \to a,\ c \to x$
(8)	$a \to \{x, y\},\ p \to b,\ c \to x$
(9)	$a \to \{x, y\},\ p \to b,\ c \to x,\ d \to \{x, y\}$
(10)	$a \to \{x, y\},\ p \to b,\ c \to x,\ d \to \{x, y\},\ b \to z$

A flow-insensitive points-to analysis determines the following information:

$$\begin{array}{l} p \to \{a,\ b\} \\ a \to \{w,\ x,\ y,\ z\} \\ b \to \{y,\ z\} \\ c \to \{w,\ x,\ y,\ z\} \\ d \to \{w,\ x,\ y,\ z\} \end{array}$$

[1]When we say that some fact is true *at* a particular point, we mean that the fact is true immediately before that point.

$a = \&w$	$a_0 = \&w_0$	(1)
$p = \&a$	$p_0 = \&a_0$	(2)
$a = \&x$	$a_1 = \&x_0$	(3)
$c = *p$	$c_0 = *p_0$	(4)
if (...)	if (...)	(5)
then $*p = \&y$	then $*p_0 = \&y_0$	(6)
$p = \&b$	$p_1 = \&b_0$	(7)
$d = a$	$d_0 = a_1$	(8)
$*p = \&z$	$*p_1 = \&z_0$	(9)
print($*a$)	print($*a_1$)	(10)
(a) Original	(b) Naive SSA	

Figure 2: An example with pointers and its naive translation to SSA form

As before, the flow-insensitive analysis is not as precise as the flow-sensitive analysis, but the information it does provide is *safe* (*i.e.,* the points-to sets computed by the flow-insensitive analysis are always supersets of the sets computed by the flow-sensitive analysis).

Given the advantages of SSA form discussed above in Section 1.2, it is natural to ask whether the approach of translating the program to SSA form and then using a flow-insensitive points-to analysis on the SSA form will achieve the same results as a flow-sensitive analysis on the original program. This approach seems reasonable since each variable x_k in the SSA form of the program corresponds only to *certain* instances of the variable x in the original program. Therefore, the 'whole-program' results of the flow-insensitive analysis of the SSA form could be mapped to 'CFG node specific' results in the original program. Unfortunately, this approach will not work.

The basic problem is that it is not possible to translate a program that contains pointers into SSA form without first doing some pointer analysis. For example, Figure 2(b) shows a naive translation to SSA form of the program shown in Figure 2(a). There are several problems with the naive translation. One problem is how the address-of operator (&) is handled. For example, in Figure 2(a) at line (2), p_0 is given the address of a_0. Clearly this is incorrect since it leads to the incorrect inference that the dereference of p_0 at line (4) is a use of a_0, when in fact it is a use of a_1, defined at line (3).

Another problem is that when a variable is defined indirectly via a pointer dereference, that definition is not taken into account in (naively) converting the program to SSA form. For example, at (6) the assignment to $*p$ is a definition of a (since at that point p contains the address of a). However, since variable a does not appear textually on the left-hand side of the assignment, the naive conversion to SSA form does not take this into account. The result is that the program in Figure 2(b) violates the first property of SSA form: that each variable x_i have exactly one definition site. Furthermore, because there is an (indirect) assignment at line (6), the use of a_1 at line (8) is reached by two definitions, thus violating the second property of SSA form.

Nevertheless, we believe that SSA form can be used to improve the results of flow-insensitive pointer analysis. An algorithm based on this idea is described below. The algorithm is iterative: it starts with purely flow-insensitive points-to information, and on each iteration it produces better information (*i.e.,* smaller points-to sets). We conjecture that when the algorithm reaches a fixed point (the last iteration produces the same points-to sets as the previous iteration) the final results mapped back to the original program will be the same as the results produced by a single run of a flow-*sensitive* pointer analysis algorithm.

Empirical studies are needed to determine how the time and space requirements of the iterative algorithm compare with those of a flow-sensitive algorithm. However, since the results of *every* iteration are *safe* (the points-to sets computed after each iteration are supersets of the actual points-to sets) the algorithm can also be safely terminated *before* a fixed point is reached (for example, after a fixed number of iterations, or when two consecutive iterations produce results that are sufficiently similar). This means that the algorithm can be 'tuned' to produce results that fall along the spectrum from flow-insensitive to flow-sensitive analysis.

2 Algorithm description

The main insight behind the algorithm is that we can use the results of (flow-insensitive) pointer analysis to normalize a program, producing an intermediate form that has two properties:

1. There are no pointer dereferences.
2. The points-to sets of all variables in the intermediate form are safe approximations to (*i.e.,* are supersets of) the points-to sets of all the variables in the original program.

Property 1 means that the intermediate form can be translated into SSA form. Property 2 means that flow-insensitive pointer analysis on the SSA form produces results that are valid for the original program.

When flow-insensitive pointer analysis is done on the SSA form, the results are in terms of the SSA variables. However, each SSA variable x_i corresponds to certain instances of the variable x in the original program. This means that the points-to set for each x_i can be mapped back to those instances of x in the original program that correspond to x_i. Note that in doing this we are producing points-to results that are no longer flow-insensitive, *i.e.,* a variable x may now have different points-to sets at different places in the program. This results in points-to sets that are often more precise than the sets produced by the initial analysis (done on the original non-SSA form of the program). These improved points-to sets can then be used to (re)normalize the program, producing a new intermediate form. If the new intermediate form is different from the previous one, the process of converting the intermediate form to SSA form, doing pointer analysis, and renormalizing can be repeated until a fixed point is reached (no change is made to the intermediate form).

Figure 3 gives an overview of the algorithm. Initially, we will assume that the input program consists of a single function with no function calls. In Section 2.1 we describe how to handle programs with multiple functions and functions calls.

The algorithm first applies flow-insensitive pointer analysis to the CFG, then uses the results to annotate each pointer dereference in the CFG with its points-to set. Only pointer dereferences are annotated because the places where we are ultimately interested in knowing about points-to information are the places where pointers are dereferenced. Note that the CFG itself is never changed, except for the annotations.

```
Given: a CFG G
Do flow-insensitive pointer analysis on G
Annotate the dereferences in G
Repeat:
    Create the intermediate form (IM) from G
    Convert IM to SSA form creating IM_SSA
    Do flow-insensitive pointer analysis on IM_SSA
    Update the annotations in G using IM_SSA
        and the pointer analysis results
until there are no changes in the annotations
```

Figure 3: An overview of the algorithm

Example: Figure 4(a) gives an example in which each pointer dereference has been annotated using the results of flow-insensitive points-to analysis. The annotations are shown to the right of each node containing a pointer dereference. For comparison, note that a flow-sensitive points-to analysis would determine that at the dereference of t, $t \to s$ and at the dereference of s, $s \to q$. $\Box$

The main loop of the algorithm begins by using the annotated CFG to create the (normalized) intermediate form (IM). In the intermediate form, each pointer dereference is replaced with its points-to set. If the points-to set contains more than one element, the single original statement is replaced with a multiway branch in which the k^{th} arm of the branch contains a copy of the original statement with the pointer dereference replaced by the k^{th} element of the points-to set. If the points-to set contains only one element, then rather than creating a branch, the pointer dereference is just replaced with the element in the points-to set.

The intermediate form is then converted to SSA form in two phases. In the first phase, conversion to SSA form is done as usual (ϕ nodes are added and variables are renamed via subscripting) with the exception that the operands of the address-of operator are not given subscripts, *i.e.,* an assignment of the form $p = \&x$ is converted to $p_i = \&x$; all other (non-address-of) uses and definitions of x are subscripted. In the second phase, each assignment of the form $p_i = \&x$ is converted to a multiway branch. The number of arms of the branch is the number of subscripts that x has in the SSA form. The k^{th} arm of the branch is of the form $p_i = \&x_k$. (As in the translation to intermediate form, if x only has one subscript, we just replace $\&x$ with $\&x_0$.)

The purpose of the second phase is to handle the first problem with translating a program with pointers to SSA form discussed in Section 1.3. Since a pointer that is given the address of x could be pointing to *any* of the SSA versions of x, using all possible versions in place of the address-of expression is a safe translation. Note that because we have replaced each pointer dereference with its points-to set we no longer have the problems mentioned in Section 1.3 that arise from the indirect definition of variables through pointer dereferences. Note also that after the second phase, the intermediate form may not be strictly in SSA form because the transformation of $p_i = \&x$ may result in multiple assignments to p_i. However, this will not affect the pointer analysis (which is the only way in which we are using this form). An equivalent way to handle $p_i = \&x$ would be to convert it as described, followed by inserting a ϕ node, and renaming the p_i's in the arms of the branch. In either case, the net result is that the definition of p that is live immediately after the transformation of $p_i = \&x$ has all SSA versions of x in its points-to set.

Example: Figure 4(b) shows the intermediate form for Figure 4(a). The intermediate form after the first phase in the conversion to SSA form is shown in Figure 4(c) and Figure 4(d) shows the final SSA form. $\Box$

The next step is to do flow-insensitive points-to analysis on the SSA version of the intermediate form (which we denote by IM_{SSA}). The results of this pointer analysis are then used to update the annotations in the CFG as follows: For each CFG node N with a pointer dereference $*p$:

- Find the corresponding node N' in IM_{SSA}. (If the node has been converted to a multiway branch construct, the branch node is the corresponding node.) Recall that all pointer dereferences were replaced with their corresponding points-to sets during the creation of the intermediate form and thus the dereferenced variable p itself is not present in N'.
- Determine the SSA number k that p would have had at node N' if it appeared there.
- Use the points-to set for p_k to update the annotation of $*p$ in node N of the CFG.

The updating of the annotations completes one iteration of the algorithm.

Example: Points-to analysis on IM_{SSA} (Figure 4(d)) determines that:

$$s_0 \to p$$
$$s_1 \to q$$
$$t_0 \to s$$

Note that because of the way the address-of operator is handled, if x_i is in p_k's points-to set, then x_j is in p_k's points-to set for all $j \in \{0, 1, 2, \ldots, \text{max\_SSA\_\#}(x)\}$. Thus, the points-to sets can be represented in canonical form by using variables without subscripts.

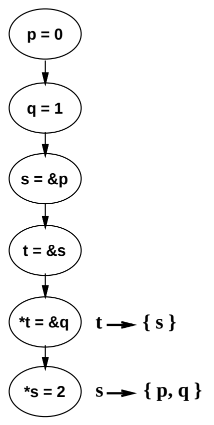
The node in IM_{SSA} that corresponds to the node $*s = 2$ in the original CFG (Figure 4(a)) is the branch node that has $p_1 = 2$ and $q_1 = 2$ as its arms. The SSA number that s would have been given if it had appeared in that branch node is 1. Because the analysis has determined that $s_1 \to q$, the node $*s = 2$ in the original CFG is annotated with $s \to \{q\}$. Figure 5(a) shows the original program with updated annotations. $\Box$

Once the annotations have been updated, the process of creating an intermediate form, converting it to SSA form, doing pointer analysis, and obtaining better annotations can be repeated. Notice that if the annotations are the same for two different iterations, then the intermediate forms created using the annotations will be identical. Thus, when no annotations are changed during the updating stage of the algorithm (*i.e.,* the annotations are the same for two successive iterations), the algorithm has reached a fixed point (*i.e.,* no new pointer information can be discovered) and the algorithm halts. Since the results of *every* iteration are *safe*, the algorithm may also be halted after a user-specified number of iterations (just after updating the annotations), resulting in pointer information that lies somewhere in between the results from a purely flow-insensitive analysis and the results had the algorithm been run to completion.

Example: Figure 5(a) shows the CFG with its annotations updated using the results of the first iteration. Figures 5(b), (c), and (d) illustrate the start of the second iteration (the intermediate form and the two-phase conversion to SSA form). Points-to analysis on Figure 5(d) determines that:

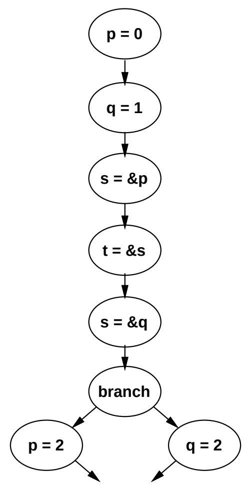
$$s_0 \to p$$
$$s_1 \to q$$
$$t_0 \to s$$

Original CFG
(with annotations)



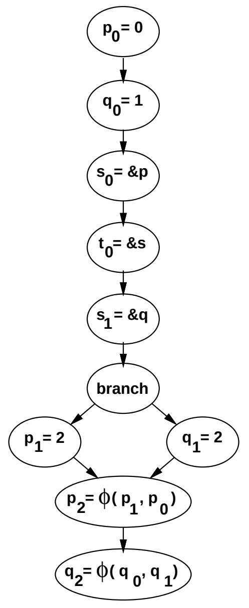
(a)

Intermediate (Normalized) Form
(*t replaced by s, and
*s replaced by p and q)



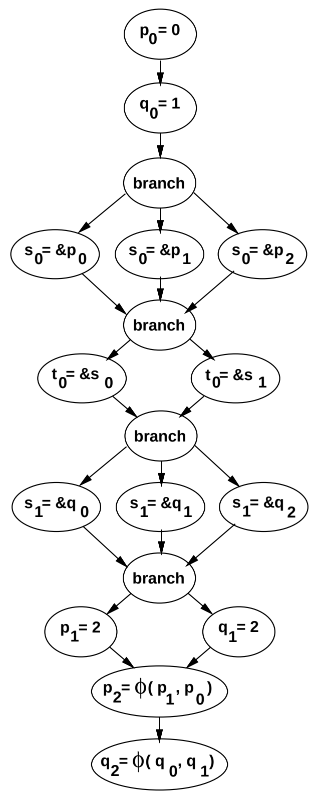
(b)

To SSA Form Phase 1
(ϕ nodes and subscripts added)



(c)

Final SSA Form
(instances of & handled)



(d)

Figure 4: Translation to SSA form (first iteration)

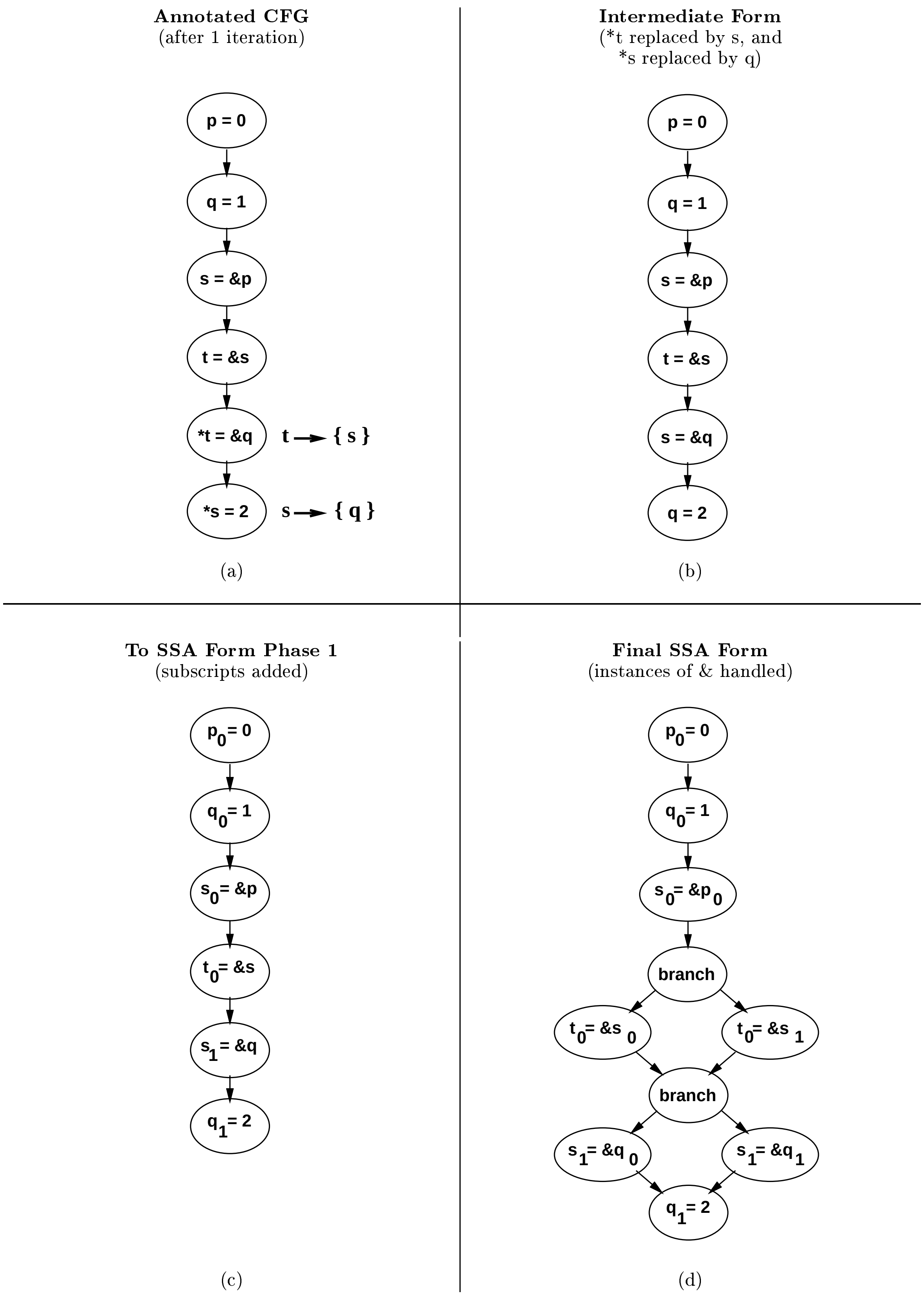


Figure 5: Translation to SSA form (second iteration)

```
int g, h;

void f()
{
  h = g;
  g = 0;
}

void main()
{
  int i;
  g = 3;
  f();
  i = g;
  if (...)
    g = 4;
  else
    f();
}
```

Figure 6: A program with multiple functions

This is the same as the information determined by the first iteration; thus, the CFG annotations do not change and the algorithm terminates after the second iteration. Note that the final results are the same as the flow-sensitive analysis on the original program. □

2.1 Handling multiple functions and function calls

A program that contains multiple functions can be represented by a set of CFGs, one for each function. However, there are problems with translating functions represented this way to SSA form when the program includes global variables. Figure 6 shows an example C program that illustrates two problems that arise when global variables are present.

One problem arises because a global variable may be used in a function before any definition of it appears in that function. For example, in the function f, global variable g is used in the assignment to h before any assignment to g. The difficulty is in determining the SSA number to give such a use. Another problem is that, because a function can modify a global variable, a use of a global variable that appears after a call may not be reached by the definition before the call. For example, in the function *main*, the value of g in the assignment to i is 0 (from the assignment $g = 0$ in f) and not 3 (from the assignment $g = 3$ before the call to f) and hence the g in $i = g$ should not have the same SSA number as the g in $g = 3$.

One way that these problems could be handled is to pass the global variables used or modified by a function as explicit parameters, and to treat the function call as an assignment to all of the global variables modified by the function.

A simpler approach is to create a supergraph[2] from the set of CFGs. The supergraph contains all nodes and edges of the original CFGs, including a call node and a return-point node for each function call. Additional edges are added from each call node to the entry node of the called function, and from the exit node of the called function to the call's return-point node. Figure 7(a) shows the CFGs for the program in Figure 6, and Figure 7(b) shows the corresponding supergraph.

[2]The term *supergraph* was first used by Eugene Myers in [Mye81]. William Landi and Barbara Ryder [LR91] use the term *interprocedural control flow graph* (ICFG).

```
Given: a list L of CFGs
Do flow-insensitive pointer analysis on L
For each CFG G in L
    Annotate the dereferences in G
Create the supergraph S for L
Repeat:
    Create the intermediate form (IM) from S
    Convert IM to SSA form creating $IM_{SSA}$
    Do flow-insensitive pointer analysis on $IM_{SSA}$
    For each CFG G in L
        Update the annotations in G using
        $IM_{SSA}$ and the pointer analysis results;
        update calls through function pointers in S
until there are no changes in the annotations
```

Figure 8: The algorithm updated to handle multiple CFGs

Calls through function pointers are represented using a multiway branch in which the k^{th} arm of the branch contains a call to the k^{th} element of the function pointer's points-to set. This requires that pointer analysis be done before the supergraph is created. Moreover, the points-to sets for function pointers may change (*i.e.,* get smaller) during iteration, so the supergraph may need to be updated (by removing some of the arms of the multi-way branches that represent calls through function pointers) when annotations are changed. Figure 8 gives the algorithm from Figure 3 updated to handle multiple CFGs.

2.2 Complexity

Each iteration of our algorithm requires a transformation to SSA form and a flow-insensitive pointer analysis.

Although there exists a linear-time algorithm for placing ϕ nodes [SG95], the renaming phase of translation to SSA form can take cubic time in the worst case. Thus, in the worst case, the time needed to completely translate a program into SSA form (including renaming) is cubic. Moreover, the resulting program can be quadratic in the size of the original program. However, experimental evidence suggests that both the time to translate and the size of the translated program are linear in practice [CFR+91] [CC95].

Andersen [And94] gives a flow-insensitive pointer-analysis algorithm that computes the greatest fixed point of the set of equations (2) given in Section 1.1. Andersen's algorithm is cubic in the worst case. Experimental evidence intended to evaluate the algorithm's performance in practice[SH97] is inconclusive: on small programs (up to about 10,000 lines) its performance is very similar to that of Steensgaard's (essentially) linear-time algorithm[Ste96]; however, lines of code alone does not seem to be a good predictor of runtime (for example, one 6,000 line program required over 700 CPU seconds, while several 7,000 line programs required only 3 seconds). Note that our algorithm could make use of a fast algorithm like Steensgaard's. However, Steensgaard's algorithm does *not* always compute the greatest fixed point of the set of equations (2). Therefore, while the final result produced by our algorithm would still be an improvement over a purely flow-insensitive analysis, it is unlikely that it would be as good as a flow-sensitive analysis that computes the greatest fixed point of the set of equations (1).

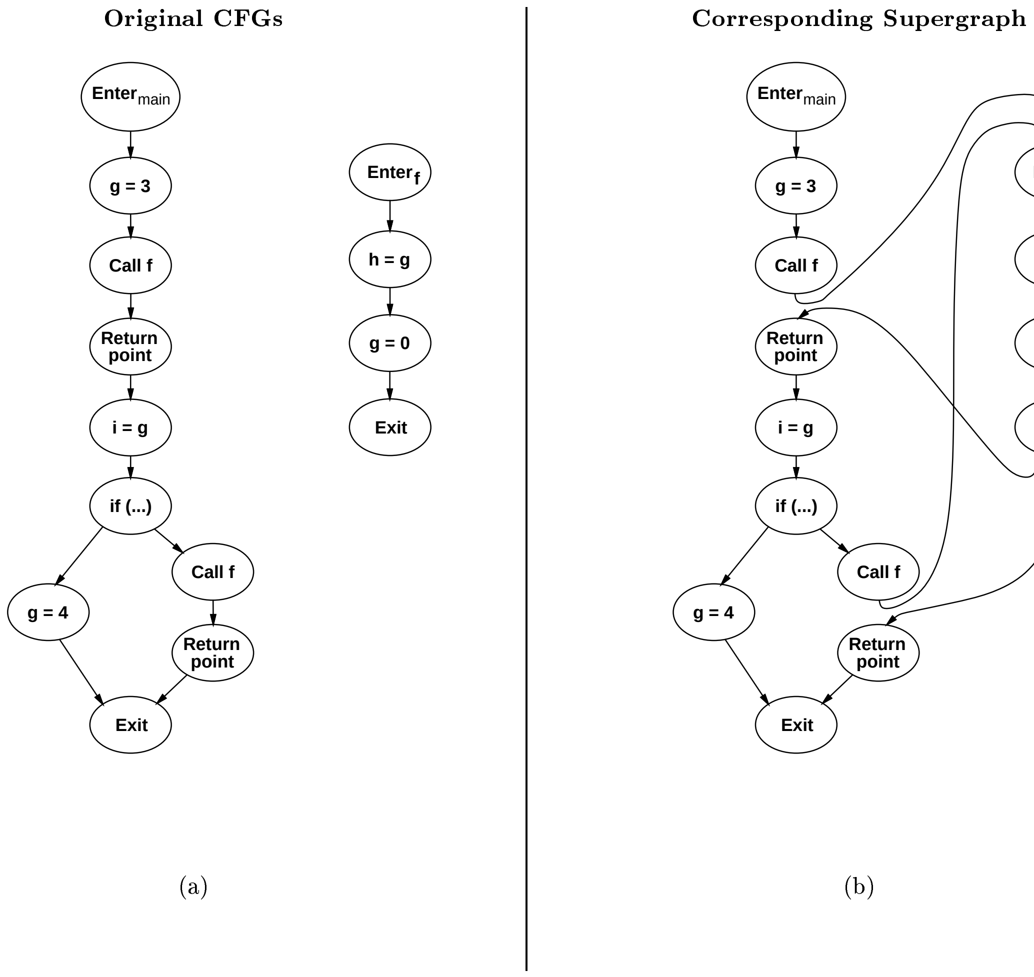


Figure 7: The CFGs and supergraph corresponding to the code in Figure 6

3 Related Work

A program representation similar to the intermediate form described here was used by Cytron and Gershbein in [CG93], where they give an algorithm for incrementally incorporating points-to information into SSA form. Our intermediate representation is essentially an in-lined version of Cytron and Gershbein's *IsAlias* function. However, their algorithm requires pre-computed may-alias information and incorporates points-to information as needed into a partial SSA form while solving another dataflow problem (constant propagation, in their paper).

Lapkowski and Hendren [LH96] also discuss the problems with SSA form in the presence of pointers. However, they abandon SSA form and develop instead a related analysis called SSA Numbering.

Others have worked on improving the precision of flow-insensitive alias analysis. In [BCCH94] Burke et al. develop an approach that involves using pre-computed *kill* information, although an empirical study by Hind and Pioli [HP97] does not show it to be more precise in practice than a flow-insensitive analysis. Shapiro and Horwitz [SH97] give an algorithm that can be 'tuned' so that its precision as well as worst-case time and space requirements range from those of Steensgaard's (almost linear, less precise flow-insensitive) algorithm to those of Andersen's (cubic worst-case but more precise flow-insensitive) algorithm.

4 Conclusions

We have presented a new iterative points-to analysis algorithm that uses flow-insensitive pointer analysis, a normalized intermediate form, and translation to SSA form. The results after just one iteration are generally better than those of a purely flow-insensitive analysis (on the original program) and if the algorithm is run until the fixed point is reached, the results may be as good as those of a flow-sensitive analysis.

We are currently working on implementations of our algorithm using the flow-insensitive pointer analyses defined in [And94], [Ste96], and [SH97]. We plan to use the implementations to explore how our algorithm compares to flow-sensitive points-to analysis in practice.

5 Acknowledgement

Thanks to Charles Consel, whose question about using SSA form in pointer analysis inspired this work.

References

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[BCCH94] M. Burke, P. Carini, J.D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Galernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250, Ithaca, NY, August 1994. Springer-Verlag.

[CC95] C. Click and K.D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[CFR$^+$91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG93] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. *SIGPLAN Conference on Programming Language Design and Implementation*, 28(6):36–45, June 1993.

[Hor97] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[HP97] M. Hind and A. Pioli. An empirical comparison of interprocedural pointer alias analyses. IBM Research Report RC 21058, IBM Research Division, December 1997.

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, January 1973.

[LH96] C. Lapkowski and L.J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. ACAPS Technical Memo 102, School of Computer Science, McGill University, Montréal, Canada, April 1996.

[LR91] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[Mye81] E.W. Myers. A precise inter-procedural data flow algorithm. In *ACM Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[SG95] V.C. Sreedhar and G.R. Gao. A linear time algorithm for placing ϕ-nodes. In *ACM Symposium on Principles of Programming Languages*, pages 62–73, 1995.

[SH97] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

Edge Profiling versus Path Profiling: The Showdown

Thomas Ball, Peter Mataga
Bell Laboratories
Lucent Technologies
{tball,mataga}@research.bell-labs.com

Mooly Sagiv*
Dept. of Computer Science
Tel-Aviv University
sagiv@math.tau.ac.il

Abstract

Edge profiles are the traditional control flow profile of choice for profile-directed compilation. They have been the basis of path-based optimizations that select "hot" paths, even though edge profiles contain strictly less information than path profiles. Recent work on path profiling has suggested that path profiles are superior to edge profiles in practice.

We present theoretic and algorithmic results that may be used to determine when an edge profile is a good predictor of hot paths (and what those hot paths are) and when it is a poor predictor. Our algorithms efficiently compute sets of *definitely* and *potentially* hot paths in a graph annotated with an edge profile. A definitely hot path has a frequency greater than some non-zero lower bound in all path profiles that induce a given edge profile.

Experiments on the SPEC95 benchmarks show that a huge percentage of the execution frequency in these programs is dominated by definitely hot paths (on average, 84% for FORTRAN benchmarks and 76% for C benchmarks). We also show that various hot path selection algorithms based on edge profiles work extremely well in most cases, but that path profiling is needed in some cases. These results indicate the usefulness of our algorithms for characterizing edge profiles and selecting hot paths.

1 Introduction

Profile-directed compilation uses run-time information gathered from one or more executions in order to better optimize programs [CMH91]. Most profile-directed optimizations depend on control flow profiles, which can be gathered via code instrumentation [BL94] or statistical sampling of the program counter [GKM83, ABD^{+}97]. There are three basic types of control flow profiles: a vertex (basic block) profile counts how many times each basic block executes during a run [KS73]; an edge (single branch) profile measures how many times each branch transition executes [BL94]; a path (correlated branch) profile measures how many times each path (of multiple branch transitions) executes.[1]

*This work was done while visiting the Software Production Research Department of Bell Laboratories.

To appear in the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1998, San Diego, CA.

Edge profiles are the traditional control flow profile for use in profile-directed compilation. They have been the basis of path-based optimizations that select heavily executed (hot) paths [Fis81, CmWH88], even though edge profiles contain strictly less information than path profiles. Recent work on path profiling has suggested that path profiles are superior to edge profiles in practice [BL96, ABL97].

We show how to determine when an edge profile is a good predictor of hot paths and when it is a poor predictor, and how to select hot paths from edge profiles. We present dynamic programming algorithms to address these questions. By focusing on directed acyclic graphs (and thus acyclic paths), we are able to create specialized algorithms that are more efficient and provide greater information than other general analysis techniques such as linear programming [Dan63] and network flow algorithms [Tar83] that can be brought to bear on the problem of predicting hot paths.

Let us consider how an edge profile constrains the possible path profiles that could induce it. Figure 1 shows four copies of a control flow graph representing a chain of three if-then-else conditionals. There are eight paths from vertex v_1 to vertex v_4 in the graph. A string of three characters (over the alphabet $\{ l, r \}$, which distinguishes a left branch from a right branch) denotes one of the eight paths. For example, the path that takes the left side of each branch is lll. The graph may represent the body of a loop or procedure, and so may be executed multiple times. In each of the four cases, the graph has been executed 100 times from vertex v_1 to v_4.

Each graph is labeled with a different edge profile (each profile is identical to the one to its immediate left, except for the boxed values). Edge frequencies in graph g_1 are skewed to the left side of branches, while edge frequencies in g_4 are much more evenly distributed. Graph g_1's edge profile greatly constrains which of the eight paths could have contributed substantially to the 100 units of flow through the graph. The path lll is always a hot path for this edge profile. Path profiling is not needed to determine this; the edge profile is sufficient.

On the other extreme, g_4's edge profile places few constraints on the eight paths: it is possible for each of the eight paths to make an approximately equal contribution to the total flow of 100, or for only two paths to create a flow near 100. In this case, path profiling is needed to determine

[1]Since there can be an unbounded number of paths for general graphs and an exponential number for directed acyclic graphs, the length of paths that are profiled must be restricted.[BL96]

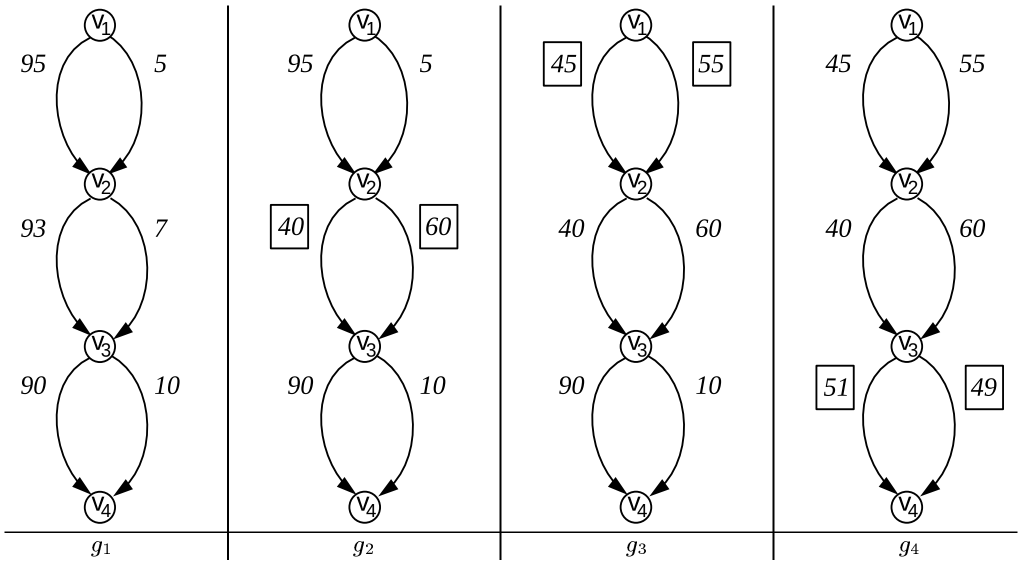


Figure 1: Four identical control-flow graphs with varying edge profiles.

which paths substantially contribute to the flow. The profiles for graphs g_2 and g_3 fall in between these two extremes. For example, in g_2, the paths *lll* and *lrl* must contribute most of the 100 units of flow. However, in g_3, the number of potentially hot paths ranges from two to four.

To make the above informal analysis a bit more precise, consider the answers to the following questions. For a given graph g_i with an edge profile (from Figure 1) and a frequency f, how many unique paths are there (from vertex v_1 to vertex v_4):

- whose frequency *definitely must* be at least f in all path profiles that induce the given edge profile?
- whose frequency *potentially could* be at least f in some path profile that induces the given edge profile?

The functions represented in Figure 2 answer the above two questions, for any frequency. For each graph g_i, there are two functions: a definitely hot path function (top), which maps a frequency f to the number of definitely hot paths from v_1 to v_4 with frequency at least f, and a similarly defined potentially hot path function (bottom). The x-axis represents frequency (ranging from 0 to 100), while the y-axis represents the value of the given function. Graph g_1 has one path with maximal definite frequency of 78, while g_4 has no paths with non-zero definite frequency.

The impact of these functions is clear. In graph g_1, 78 out of 100 units of flow definitely can be assigned to a single path (*lll*). In graph g_2, 70 units of flow are definite, contributed by the paths *lrl* (45) and *lll* (25). However, in g_3 there is a very small amount of definite flow (5 units), while in g_4 there is none. As the amount of definite flow decreases, the number of paths with higher potential frequency increases, indicating that there is a great deal of variability in how frequencies may be assigned to paths. In these cases, path profiling will be need to determine the hot paths.

Our results are three-fold:

- We provide constructive characterizations of the definitely and potentially hot path functions and show that both can be efficiently encoded using a multiset whose cardinality is linear in the number of edges in the input graph. These observations lead straightforwardly to efficient dynamic programming algorithms for computing definitely and potentially hot path functions from edge profiles, as well as to an algorithm for enumerating these paths in decreasing order of frequency.
- We show how to determine when an edge profile is a good approximation to a path profile, such as in graphs g_1 and g_2, and when path profiling is necessary, such as in graphs g_3 and g_4. The fundamental insight is that the integral of the definitely hot path function is an exact lower bound on the total flow through a procedure. Thus, if a procedure (such as g_1 or g_2) has a definite flow integral close to the total flow, much of the flow can be assigned to a fixed set of paths.
- Experimental results comparing edge profiles and path profiles from the SPEC95 benchmarks. These results show that, on average, 84% of the total flow in the FORTRAN benchmarks is definite flow, while 76% of the flow in the C benchmarks is definite, both remarkably high values. This large amount of definite flow means that hot path predictors based on edge profiles can do very well, which we also show independently by comparing our and other hot path predictors to paths computed by the PP path profiling tool [BL96].

This paper is organized as follows. Section 2 discusses some of the tradeoffs in profiling programs and highlights results in edge and path profiling to date. Section 3 defines definitely and potentially hot paths and describes the dynamic programming algorithms for computing and enumerating these paths from edge profiles. Section 4 applies our algorithms to edge profiles collected from the SPEC95

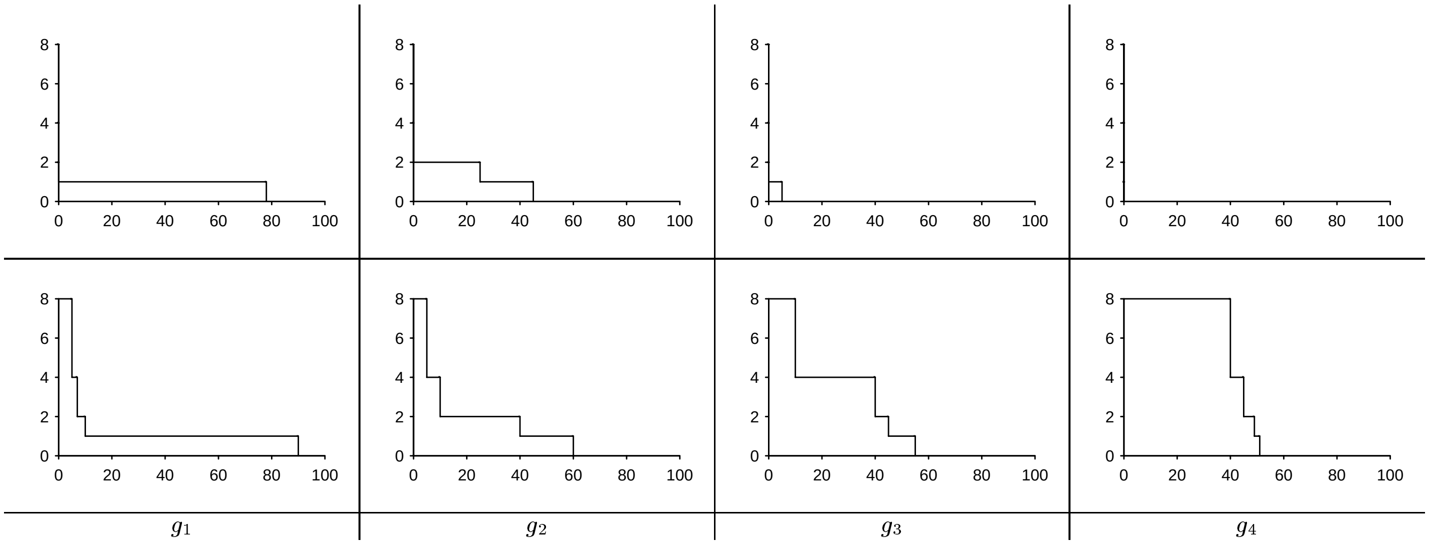


Figure 2: The definitely and potentially hot path functions for the four graphs from Figure 1. For each graph (g_1 to g_4), there are two functions plotted: the definitely hot path function (top), and the potentially hot path function (bottom). The x-axis represents frequency ($0 \dots 100$) and the y-axis is the number of paths that definitely (potentially) have that frequency.

benchmarks and compares the results to path profiles obtained via a path profiling tool. Section 5 discusses related work and Section 6 concludes the paper.

2 Edge Profiles vs. Path Profiles: Historical Context

There are two main issues in choosing what type of profile to collect for optimization: the run-time overhead incurred in order to collect the profile, and the accuracy and quality of the collected data required for the intended optimization.

Vertex and edge profiles are relatively cheap to collect via instrumentation [KS73, BL94], incurring around 16% overhead. Two recent results have improved the state of profiling further: researchers at DEC used a modified operating system kernel to efficiently collect vertex profiles by sampling the program counter, incurring only 1-3% overhead [ABD$^+$97]; Ball and Larus developed a simple instrumentation technique for path profiling that efficiently collects frequency information for acyclic paths in a procedure [BL96, ABL97], with an overhead of about 30%.

Theoretically, path profiles contain strictly more information than edge profiles. A precise edge profile can be derived from a path profile (as shown in the next section), but the converse does not hold. Stated another way, there are many different path profiles that induce the same edge profile, but different edge profiles imply different path profiles. Thus, path profiles provide the most information about control flow behavior, subsumed only by the profiling of longer and longer paths. The ultimate form of a path profile is a trace of an entire execution, which is extremely costly to collect and store.

Today, edge profiles are the control flow profile of choice for compiler optimization. We identify three main reasons for this. First, edge profiles provide a clear advantage over vertex profiles as they provide more accurate information about the behavior of branches, which is crucial to many optimizations. Second, edge profiles are generally thought to be easier and cheaper to collect than path profiles (although the Ball-Larus work has shown that path profiles may be collected with relatively low overhead). Third, there is a good amount of literature that indirectly suggests that edge profiles are an acceptable substitute for path profiles. In particular, work on branch prediction and trace-based optimizations has implicitly addressed the power of edge profiles to predict hot paths:

- Branch prediction studies of the SPEC benchmarks and other programs [FF92, BL93] have shown that many branches are well-behaved in the following sense: they take one direction with high probability and this high probability direction remains the same over different executions.[2] The implication for path profiling is clear: a succession of highly skewed branches implies that one path must execute with much greater frequency than the remaining paths (e.g., graph g_1 in Figure 1).

- Trace-driven optimization has long used edge profiles to predict hot paths [Fis81, CMH91]. A greedy algorithm is typically applied: starting at a hot vertex or edge, the path is grown backwards or forwards by choosing an edge incident to the beginning or end of the path with maximal execution frequency. For example, the Impact compiler uses a set of such heuristics to partition the control flow graph into a set of non-overlapping paths [CmWH88]. However, the foundational basis for such algorithms is weak and the algorithms cannot predict when they will have good success.

Weighing in on the side of path profiles, there is evidence to suggest that path profiles are superior to edge profiles in practice. A number of studies on dynamic branch prediction mechanisms have shown that history-sensitive mechanisms

[2]For example, consider conditionals that perform error checks, which rarely raise exceptions.

based on correlated branches (paths, that is) improve hardware branch prediction mechanisms [Smi81, LS84, PSR92]. Ball and Larus compared the paths chosen by a greedy hot path algorithm with those measured by their path profiling tool, and found that the heuristic does indeed choose incorrectly at times [BL96].

3 Path Profiles from Edge Profiles

This section presents efficient algorithms for computing definitely and potentially hot paths from edge profiles. Section 3.1 precisely defines definitely and potentially hot paths via the functions $\hat{D}$ and $\hat{P}$. Section 3.2 constructively characterizes these functions and describes some of their essential properties. Section 3.3 presents our dynamic programming algorithm for computing the $\hat{D}$ function, and Section 3.4 does likewise for $\hat{P}$. Section 3.5 shows how to enumerate the potentially and definitely hot paths by decreasing order of frequency.

3.1 Definitely and Potentially Hot Paths

First, let us define some graph notations that are used in the sequel.

Definition 3.1 *Let $G(V, E)$ be a directed acyclic control flow graph (DAG) with a unique entry vertex "entry" from which all vertices are reachable, and a exit vertex "exit" that is reachable from all vertices. An edge $e = v \rightarrow w$ connects source vertex v (denoted by $src(e)$) to target vertex w (denoted by $tgt(e)$). Let $out(v)$ represent the edges e such that $v = src(e)$ and let $in(v)$ represent the edges e such that $v = tgt(e)$.*

A path in G is represented as a sequence of edges $p = [e_1, e_2, \ldots, e_n]$, where $tgt(e_i) = src(e_{i+1})$. Let $P(G)$ denote the set of all paths from entry to exit. The set of paths from entry to exit in which edge e appears is denoted by $P(e)$.

Next, we define properties of an edge profile *freq*.

Definition 3.2 *For an edge $e \in E$, let $freq(e)$ be the number of times that the edge e was executed. For a vertex v, $freq(v)$ is the number times v executed. An edge profile respects conservation of flow at a vertex. That is, for every $v \in V - \{entry\}$,*

$$freq(v) = \sum_{e \in in(v)} freq(e)$$

and for every $v \in V - \{exit\}$,

$$freq(v) = \sum_{e \in out(v)} freq(e)$$

It is straightforward to see that the conservation of flow implies that $freq(entry) = freq(exit)$, the total flow through the DAG, which is denoted by F.

We now define when an assignment *pfreq* of frequencies to paths is *admissible* with respect to an edge profile *freq*.

Definition 3.3 *A frequency assignment pfreq to all paths in $P(G)$ is* admissible *w.r.t. an edge profile freq if for every $e \in E$,*

$$freq(e) = \sum_{p \in P(e)} pfreq(p)$$

The above equation shows how to derive an edge profile from a path profile. Although *pfreq* assigns frequencies only directly to paths in $P(G)$, it indirectly assigns a frequency to every path in the DAG. For any path q, let $subpfreq(q)$ denote q's frequency under *pfreq*, which is simply the sum of all $pfreq(p)$ such that p is a path in $P(G)$ that contains q as a subpath. The total flow through G may be expressed in terms of paths as:

$$F = \sum_{p \in P(G)} pfreq(p)$$

With admissible path frequency assignments in hand, we now make precise the notion of definite and potential path frequencies:

Definition 3.4 *A directed path p from v to exit* has a definite frequency f *if for every admissible frequency assignment pfreq, $subpfreq(p) \geq f$.*

A directed path p from v to exit has a potential frequency f *if there exists an admissible frequency assignment pfreq in which $subpfreq(p) \geq f$.*

The number of paths from v to exit with definite (respectively, potential) frequency f is denoted by $\hat{D}[v](f)$ (respectively, $\hat{P}[v](f)$). The domain of $\hat{D}[v]$ and $\hat{P}[v]$ may be restricted to the non-negative integers in $0 \ldots F$.

Example 3.5 Figure 2 shows the functions $\hat{D}[v_1]$ (top row) and $\hat{P}[v_1]$ (bottom row) for each of the four example graphs from Figure 1. Graph g_1 has one path with maximal definite frequency of 78. That is, $\hat{D}[v_1](f) = 1$ for $f \leq 78$ and for larger frequencies f, $\hat{D}[v_1](f) = 0$. In g_4, no paths have definite frequency greater than zero. In g_2, there is one path (lrl) of definite frequency 45 and two of definite frequency 25 (lrl and lll). Since definite frequencies hold for all admissible assignments, these frequencies are simultaneously realizable. That is, in all path profiles that induce this edge profile, lrl will have a frequency of at least 45 and lll will have a frequency of at least 25.

In graph g_1, the maximum potential frequency of a path is 90 (path lll). Graph g_2 has two paths whose potential frequency can exceed 30. In contrast, in g_3, $\hat{P}[v_1](30) = 4$ since there are four paths from v_1 to v_4 whose frequency can exceed 30. Finally, in g_4, $\hat{P}[v_1](30) = 8$. It is clear that eight paths cannot simultaneously have a frequency of 30 in an admissible path frequency assignment for graph g_4. Unlike definite frequencies, potential frequencies are not necessarily simultaneously realizable.

As should now be clear, the functions $\hat{D}[v]$ and $\hat{P}[v]$ are anti-monotonic. That is, for every f_1 and f_2 such that $f_1 \leq f_2$:

$$\hat{D}[v](f_1) \geq \hat{D}[v](f_2) \text{ and } \hat{P}[v](f_1) \geq \hat{P}[v](f_2)$$

3.2 Constructive Characterizations of $\hat{D}$ and $\hat{P}$ and Properties

The definite frequency for a path p intuitively depends upon the extent to which paths with which p shares edges are able to "steal flow" from p. More precisely, for any path p that ends with vertex *exit*, define

$$B_D(p) = F - \sum_{e \in p} \sum_{e' \in join(e)} freq(e')$$

where for any edge $e \in E$

$$join(e) = \{e' \in E \mid e' \neq e \text{ and } tgt(e') = tgt(e)\}$$

The following theorem shows that this expression is exact.

Theorem 1 *A path p has a non-zero definite frequency f if and only if $f \leq B_D(p)$.*
Proof: See the appendix.

Example 3.6 Consider $B_D(lll)$ for graph g_1 in Figure 1. In g_1, the smallest frequency that the path lll can have in any admissible frequency assignment is:

$$\begin{aligned} & 100 - ((100-95) + (100-93) + (100-90)) \\ = \; & 100 - (5+7+10) = 100 - 22 \\ = \; & 78 \end{aligned}$$

which is obtained by subtracting out the frequencies of the other edges that join into lll (namely, 5, 7 and 10) from the flow at v_4 (100). On the other hand, $B_D(llr)$ is equal to:

$$\begin{aligned} & 100 - ((100-95) + (100-93) + (100-10)) \\ = \; & 100 - (5+7+90) \\ = \; & 100 - 102 < 0 \end{aligned}$$

and therefore llr has a definite frequency of zero.

A surprising result is that the number of paths with non-zero definite frequency is sub-linear in $|E|$, as proved in the theorem below. This fact, along with the anti-monotonicity of $\hat{D}[v]$ means that the function can be represented accurately and efficiently by recording the steps of the function. A step $f \mapsto \Delta$ is denoted by a pairing of a frequency f and a magnitude Δ, where

$$\Delta = \hat{D}[v](f) - \hat{D}[v](f+1) > 0$$

Theorem 2 *For all vertices v, the set of paths from v to exit with non-zero definite frequency has cardinality less than $width(G_v)$,[3] where $width(G)$ is the size of a maximal cut of G (i.e., the number of edges crossing a maximal cut, over all cuts of G, which is bounded by $|E|$).*
Proof: See the appendix for full details. The proof relies on the unique topological structure of paths with non-zero definite frequency: any two paths with non-zero definite frequency can split (or join) at most once.

Example 3.7 The width of the graph in Figure 1 is two, and as can be seen in Figure 2, the number of paths with non-zero definite frequency never exceeds two. In graph g_2, the paths lll and lrl both have non-zero definite frequency. These paths split from one another exactly once (at vertex v_2).

A path can have a potential frequency f if each of the edges in the path have enough capacity to admit f. This is captured by the following theorem:

Theorem 3 *Given an edge profile freq, a path $[e_1, e_2, \ldots, e_n]$ from v to exit has potential frequency f if and only if for every i, $f \leq freq(e_i)$.*
Proof: Trivial.

[3] G_v denotes the subgraph of G containing the vertices and edges of G that are reachable from v.

$$\hat{D}[exit] = \lambda f. \begin{cases} 1 & f \leq F \\ 0 & \text{otherwise} \end{cases}$$

$$\forall\ e,\ \hat{D}[e] = \lambda f. \hat{D}[tgt(e)](f + (freq(tgt(e)) - freq(e)))$$

$$\forall\ v, v \neq exit,\ \hat{D}[v] = \lambda f. \textstyle\sum_{e \in out(v)} \hat{D}[e](f)$$

Figure 3: Equational definition of $\hat{D}$ at every vertex v and edge e. Recall that $F = freq(exit)$.

```
$M_{\hat{D}}[exit] := [F \mapsto 1]$
**for** $v \in V - \{exit\}$ in reverse topological order **do**
    **for** $e \in out(v)$ **do**
        $f_s := freq(tgt(e)) - freq(e)$
        $M_{\hat{D}}[e] := \biguplus_{(f \mapsto \Delta) \in M_{\hat{D}}[tgt(e)]} (f > f_s \ ?\ [(f - f_s) \mapsto \Delta] : [\,]\,)$
    **od**
    $M_{\hat{D}}[v] := \biguplus_{e \in out(v)} M_{\hat{D}}[e]$
**od**
```

Figure 4: An efficient realization of $\hat{D}[v]$, using a multiset $M_{\hat{D}}[v]$.

Example 3.8 Consider path lll in graph g_2 from Figure 1. The minimum edge frequency in this path is 40. Therefore, lll's frequency can never exceed 40 in an admissible assignment. There is an admissible assignment to all paths in which lll has frequency 40, since subtracting 40 from the three edges of lll leaves an edge profile which conserves flow at each vertex.

It follows from Theorem 3 that the steps of $\hat{P}[v]$ can only occur at frequencies in the set

$$\{freq(e) \mid e \in E \text{ such that } e \text{ is reachable from } v\}$$

Therefore, even though every path in a DAG can have a maximal non-zero potential frequency (which was not the case with definite frequencies) and the number of paths in a DAG is exponential (in the worst case), the number of steps of $\hat{P}[v]$ is linear in the size of the DAG.

3.3 Definitely Hot Paths: Algorithms

An equational definition of $\hat{D}[v]$ is given in Figure 3. The number of definitely hot paths from v to *exit* is defined inductively. For the *exit* vertex and any frequency less than or equal to F, there is one path of that frequency (since there is trivially a path from *exit* to *exit* of frequency F). The function $\hat{D}[e]$ is defined solely in terms of $\hat{D}[tgt(e)]$, based on Theorem 1. The edge e shifts the frequency of the steps of the function $\hat{D}[tgt(e)]$ down by $(freq(tgt(e)) - freq(e))$, the amount of flow that paths that join at $tgt(e)$ can steal from paths containing edge e. At any vertex v other than *exit*, $\hat{D}[v]$ is simply the sum of $\hat{D}[e]$, for all e in $out(v)$.

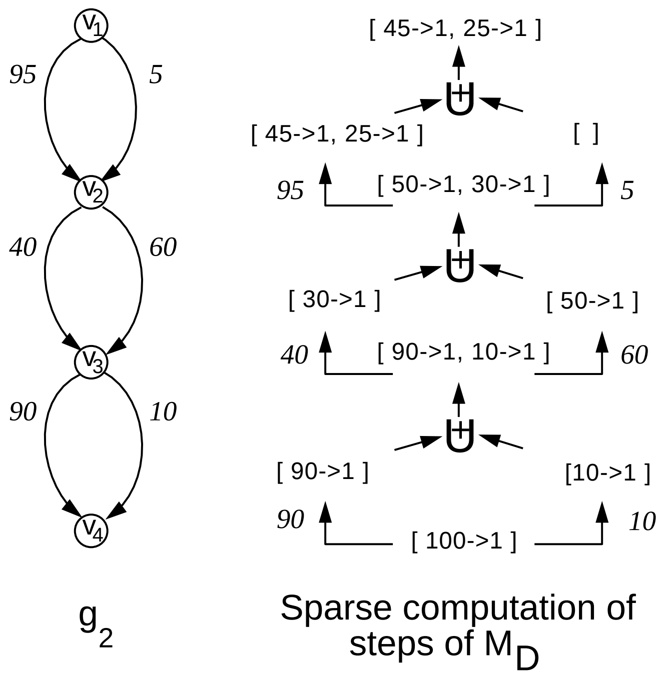


Figure 5: Results of the $M_{\hat{D}}$ algorithm from Figure 4 on graph g_2.

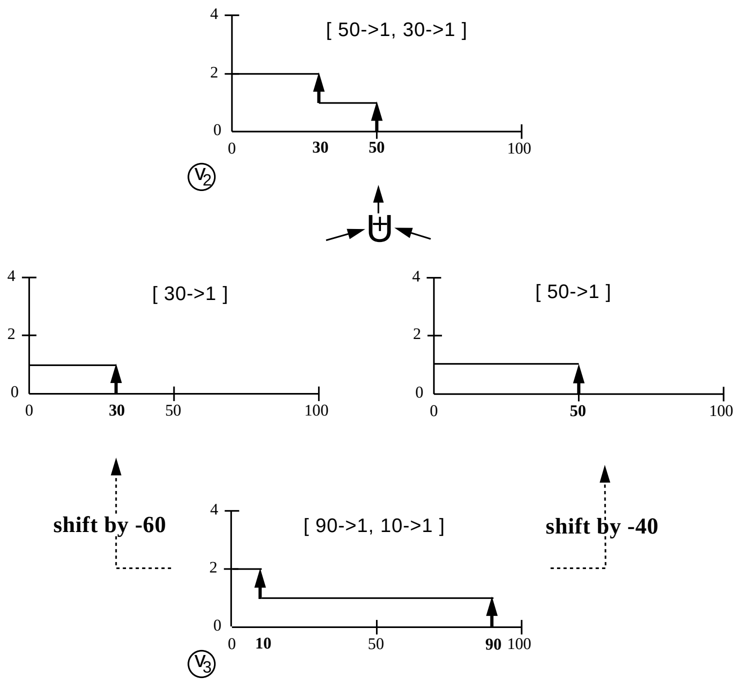


Figure 6: Illustration of the manipulation of the $M_{\hat{D}}$ map on graph g_2. See text for further explanation.

Because the graph is acyclic, the set of equations is not recursive. In fact, the dependence graph of the set of equations is isomorphic to the original DAG. The equations trivially yield a demand-driven algorithm for computing $\hat{D}[v](f)$, given a vertex v and frequency f. However, this does not yield an efficient algorithm for computing $\hat{D}[v]$ for all v.

We now provide a sparse realization of the equational definition in which the step frequencies and step magnitudes for the function $\hat{D}[v]$ are accumulated in a partial function (or multiset or map) $M_{\hat{D}}[v]$. That is,

$$(f \mapsto \Delta) \in M_{\hat{D}}[v] \quad \iff \quad \Delta = \hat{D}[v](f) - \hat{D}[v](f+1) > 0$$

so that

$$\hat{D}[v](f) = \sum_{\substack{(g \mapsto \Delta) \,\in\, M_{\hat{D}}[v] \\ g \,\geq\, f}} \Delta$$

Recall that the numbers of steps is bounded by the number of edges in the graph.

Figure 4 presents an efficient algorithm for computing $M_{\hat{D}}[v]$ for all v. The algorithm uses a multiset union operator ($\uplus$) which adds the ranges (magnitudes) of steps with identical frequencies. For example,

$$\begin{aligned}&[40 \mapsto 1, 10 \mapsto 1] \;\;\uplus\;\; [60 \mapsto 1, 10 \mapsto 1] \\ &= [40 \mapsto 1, 60 \mapsto 1, 10 \mapsto 2]\end{aligned}$$

The algorithm works as follows. The multiset for the *exit* vertex is initialized to $[F \mapsto 1]$. Vertices are then visited in reverse topological order so that a vertex v is considered only after all of its successors have been visited. The algorithm first computes $M_{\hat{D}}[e]$ for all outgoing edges e of v and then computes $M_{\hat{D}}[v]$ using these maps. To create $M_{\hat{D}}[e]$ the algorithm shifts the function $M_{\hat{D}}[tgt(e)]$ down by f_s. To do so, it considers each $(f \mapsto \Delta)$ in $M_{\hat{D}}[tgt(e)]$ and adds $(f - f_s) \mapsto \Delta$ to $M_{\hat{D}}[e]$ when f is greater than f_s.

Example 3.9 Figure 5 shows the iterations of the sparse $M_{\hat{D}}$ algorithm for the graph g_2 from Figure 1. The graph g_2 is repeated on the left. The diagram to the right of g_2 shows the maps flowing upward through the vertices and edges of the graph, where each step is represented by a pair $(f \mapsto \Delta)$.

Figure 6 illustrates how $M_{\hat{D}}[v_2]$ is computed in greater detail. The chart on the bottom simultaneously represents $\hat{D}[v_3]$ and $M_{\hat{D}}[v_3]$. The x-axis represents frequency and the y-axis represents the value of $\hat{D}[v_3]$. The steps are denoted by the bold arrows. The magnitude of a step is represented by the magnitude of the arrow. The chart on the left represents the map for the left outgoing edge of v_2, obtained by shifting $M_{\hat{D}}[v_3]$ by -60 (that is, $-(100-40)$), while the chart on the right represents the map for the right outgoing edge of v_2, obtained by shifting $M_{\hat{P}}[v_3]$ by -40. The chart on the top represents the multiset union of the two edge maps.

3.4 Potentially Hot Paths: Algorithms

An equational definition of $\hat{P}[v]$ is given in Figure 7, based on Theorem 3. Note the similarity to the definition of $\hat{D}[v]$ in Figure 3. In fact, the only change is in the definition of the edge function. In this case, the edge e masks the frequency of steps in $\hat{P}[tgt(e)]$ that are greater than $freq(e)$. That is, an edge e has $\hat{P}[e](f) = \hat{P}[tgt(e)](f)$ paths if f does not exceed $freq(e)$ and zero paths otherwise.

$$\hat{P}[exit] = \lambda f. \begin{cases} 1 & f \leq F \\ 0 & \text{otherwise} \end{cases}$$

$$\forall\ e,\ \hat{P}[e] = \lambda f. \begin{cases} \hat{P}[tgt(e)](f) & f \leq freq(e) \\ 0 & \text{otherwise} \end{cases}$$

$$\forall\ v,\ v \neq exit,\ \hat{P}[v] = \lambda f. \textstyle\sum_{e \in out(v)} \hat{P}[e](f)$$

Figure 7: Equational definitions of $\hat{P}$ at every vertex v and edge e.

```
$M_{\hat{P}}[exit] := [F \mapsto 1]$
**for** $v \in V - \{exit\}$ in reverse topological order **do**
    **for** $e \in out(v)$ **do**
        $M_{\hat{P}}[e] := \biguplus_{f \mapsto \Delta \in M_{\hat{P}}[tgt(e)]} [min(f, freq(e)) \mapsto \Delta]$
    **od**
    $M_{\hat{P}}[v] := \biguplus_{e \in out(v)} M_{\hat{P}}[e]$
**od**
```

Figure 8: An efficient realization of $\hat{P}[v]$, using multiset union.

Figure 8 presents the efficient algorithm for computing $M_{\hat{P}}[v]$, which has nearly identical structure to that for $M_{\hat{D}}[v]$. Again, the crucial difference is in the edge computation. $M_{\hat{P}}[e]$ simply masks out all elements of $M_{\hat{P}}[tgt(e)]$ with frequency greater than $freq(e)$. This is achieved by a multiset union over the elements of $M_{\hat{P}}[tgt(e)]$, using the *min* function to perform the masking. This copies through elements (from $M_{\hat{P}}[tgt(e)]$ to $M_{\hat{P}}[e]$) with frequency less than $freq(e)$ and accumulates the magnitudes of steps with frequency greater than or equal to $freq(e)$ to form the correct step magnitude at $freq(e)$.

Example 3.10 Figure 9 shows the iterations of the $M_{\hat{P}}$ algorithm for the graph g_2 from Figure 1. The graph g_2 is repeated on the left. The diagram to the right of g_2 shows the maps flowing upward through the vertices and edges of the graph.

Figure 10 illustrates how $M_{\hat{P}}[v_2]$ is computed in greater detail. As before, the charts simultaneously represent $\hat{P}$ and $M_{\hat{P}}$, with the bottom chart representing the map $M_{\hat{P}}[v_3]$. The left chart represents the map for the left outgoing edge of v_2, obtained by masking $M_{\hat{P}}[v_3]$ at frequency 40, while the chart on the right represents the map for the right outgoing edge of v_2, obtained by masking $M_{\hat{P}}[v_3]$ at frequency 60. The chart on the top represents the multiset union of the two edge maps.

3.5 Enumerating the Paths

Figure 11 presents an algorithm that enumerates the definitely or potentially hot paths in descending order of frequency, using the vertex and edge maps computed by the

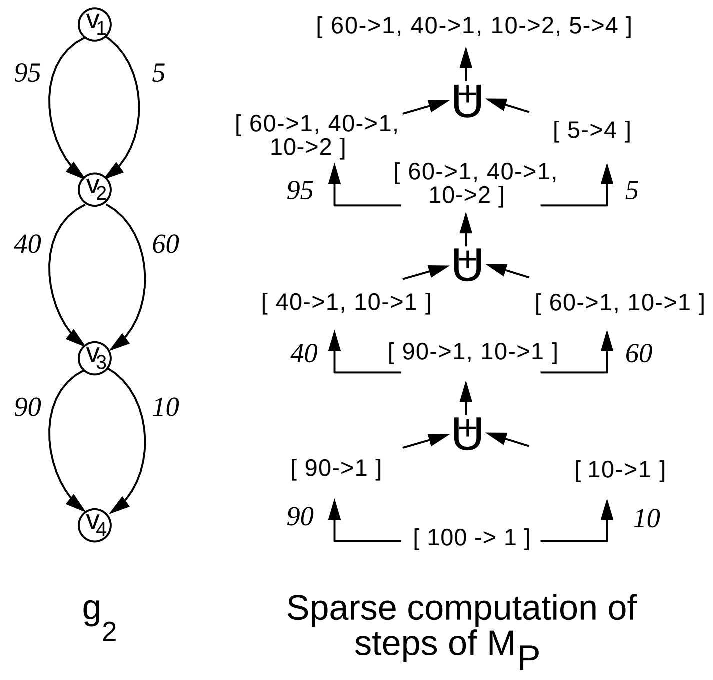


Figure 9: Results of the $M_{\hat{P}}$ algorithm from Figure 8 on graph g_2.

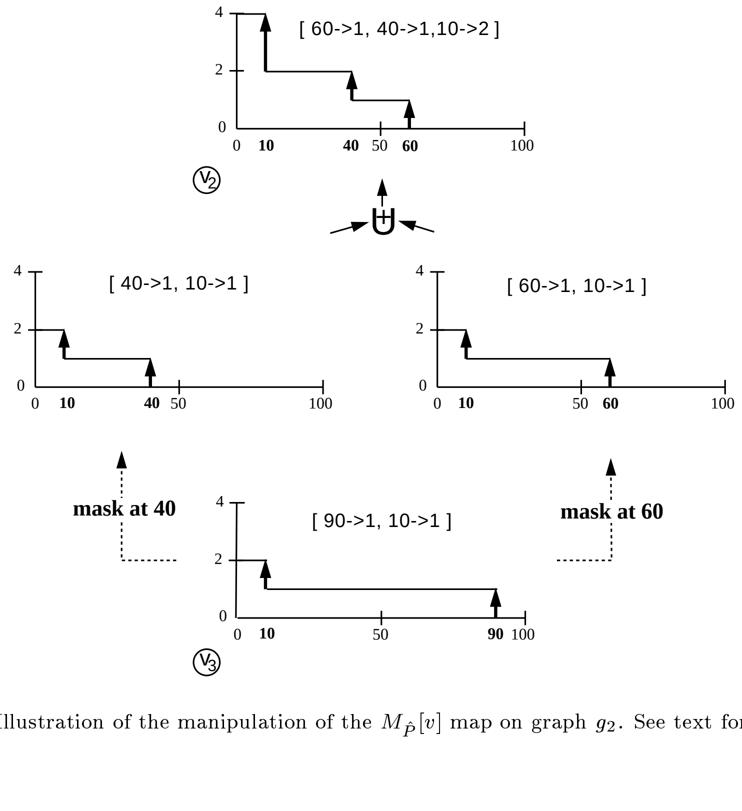


Figure 10: Illustration of the manipulation of the $M_{\hat{P}}[v]$ map on graph g_2. See text for explanation.

algorithms of the previous sections. This algorithm enumerates the paths in time proportional to the number of paths requested.

The procedure `main` accepts a map, either $M_{\hat{D}}$ or $M_{\hat{P}}$, and returns a set of the definitely or potentially hot paths (respectively) whose frequency is greater than *cutoff*. The algorithm can be easily modified to return the top n paths, rather than those meeting a frequency threshold.

To understand the operation of the algorithm, consider a call to procedure `enumerate`. This procedure is passed the current vertex v, the path prefix p constructed so far, the frequency f of the path (which is simply passed down through successive calls to `enumerate`) and Δ, the number of paths with frequency f to be enumerated from v onwards. If v is the exit vertex, then the recursion bottoms out, recording the path. Otherwise, the algorithm distributes the Δ paths onto the outgoing edges of v. The construction of the maps $M_{\hat{D}}$ and $M_{\hat{P}}$ guarantees that the Δ paths can be consumed by the outgoing edges of v.

Example 3.11 Consider the operation of the enumeration algorithm on the maps in Figure 9. Suppose that *cutoff* is 30. The algorithm will first call `enumerate` with $f = 60$ and $\Delta = 1$. The algorithm consults the maps for the outgoing edges of v_1 and finds that $(60 \mapsto 1)$ is the only step that could permit this path. So, `enumerate` is recursively called with $f = 60$ and $\Delta = 1$, appending the left outgoing edge of v_1 to the path p. At v_2 it picks the step $(60 \mapsto 1)$ of the right edge and at v_3 it picks the step $(90 \mapsto 1)$ from the left edge. Thus, the path *lrl* with frequency 45 is enumerated. The next top level call to `enumerate` will pass in $f = 40$ and $\Delta = 1$. In this case, the path *lll* will be enumerated. The left edge leaving v_2 will be chosen because 40 is the smallest frequency greater than or equal to f in the maps of edges leaving v_2.

```
procedure main($M$ : map; $cutoff$ : integer)
var $Paths := \phi$
procedure enumerate($v$ : vertex; $p$ : path; $f, \Delta$ : integer)
var $\Delta' := \Delta$
    $used := \phi$
begin
    if $v = exit$ then
        $Paths := Paths \cup \{(p, f)\}$
    else
        while $\Delta' > 0$ do
                let $e \in out(v)$ and
                    $(g \mapsto \Delta_g) \in M[e]$ s.t.
                      $g$ is minimal, $g \geq f$ and $(e, g) \notin used$
                    $debit = min(\Delta', \Delta_g)$
                in
                    enumerate($tgt(e)$, append($p, e$), $f$, $debit$)
                    $used := used \cup \{(e, g)\}$
                    $\Delta' := \Delta' - debit$
                ni
        od
    fi
end enumerate
begin
    for each $(f \mapsto \Delta) \in M[entry]$ s.t. $f \geq cutoff$,
        in decreasing order of $f$ do
            enumerate($entry$, [ ], $f$, $\Delta$)
    od
    return $Paths$
end main
```

Figure 11: A procedure to enumerate hot paths in decreasing order of frequencies.

4 Experimental Results

We used the Ball-Larus instrumentation tool (PP) to examine the prediction of hot paths from edge profiles [BL96]. PP profiles intraprocedural control flow paths that start either at procedure entry or a loop backedge and end at procedure exit or a loop backedge. We ran PP on the SPEC95 benchmarks. Each instrumented benchmark was run on one input and a path profile was collected. An edge profile was computed from the path profile by counting how many times each edge appeared in the executed paths (see Definition 3.3).

A cyclic control flow graph is transformed an acyclic graph (DAG) as follows: for each backedge $v \rightarrow w$, remove it from the graph and in its place add two edges: $entry \rightarrow v$ and $w \rightarrow exit$. The frequency of the backedge $v \rightarrow w$ in the edge profile is assigned to its replacement edges, thus conversing flow at v and w.

4.1 Benchmark Characterization

The SPEC95 benchmarks consist of eighteen programs, nine written in FORTRAN and nine written in C. Table 1 lists summary data for each benchmark: the number of paths executed (**# Paths**); the total flow for the benchmark (that is, the sum of the frequencies of all paths executed, in millions–**Total Flow**); the number of paths whose frequency is greater than 0.125% of the total flow, and the percentage of total flow these paths contribute in aggregate; the percentage of paths whose frequency is greater than one percent of the total flow, and the percentage of total flow they contribute. In most benchmarks, a relatively small number of paths (compared to the total executed) contribute to a large amount of the total flow.

4.2 Definite and Potential Flow

Our first experiment determines the amount of definite and potential flow in each benchmark from the edge profile. For each procedure, the amount of definite flow is

$$\sum_{(f \mapsto \Delta) \ \in \ M_{\hat{D}}[entry]} f * \Delta$$

which is simply the integral of the function $\hat{D}[entry]$. The amount of definite flow for the whole benchmark is the sum of the definite flow integrals for all procedures, which is less than the total flow for the benchmark. A similar computation was done using $M_{\hat{P}}[entry]$ to compute the potential flow integral for each benchmark.

Figure 12(a) presents the ratio of the definite flow integral to the total flow for each benchmark. The benchmarks are presented in two groups, the FORTRAN codes on the left and the C programs on the right. The benchmarks are sorted within each group by the ratio. Note that there is a substantial amount of definite flow in all benchmarks, even

Benchmark	Num. Paths	Total Flow (M)	0.125% Cutoff # paths	0.125% Cutoff % flow	1.0% Cutoff # paths	1.0% Cutoff % flow
applu	572	1,735.5	61	99.2	32	84.9
apsi	1,040	599.0	80	92.9	23	70.5
fpppp	521	291.9	51	98.6	17	82.9
hydro2d	1,033	599.9	40	97.4	27	92.1
mgrid	570	994.8	32	98.7	4	89.0
su2cor	837	530.1	64	99.0	22	87.4
swim	370	163.3	13	98.8	3	95.9
tomcatv	407	375.7	23	96.1	5	92.9
turb3d	667	2,952.7	70	98.4	19	77.5
compress	248	3,015.7	41	99.6	19	91.1
gcc	9,375	9.7	66	64.3	5	48.7
go	11,671	779.7	105	75.2	21	49.3
ijpeg	1,068	1,155.9	48	95.5	16	85.7
li	796	3,357.9	70	97.0	32	81.7
m88ksim	1,053	4,772.4	61	95.4	15	81.6
perl	1,339	1,137.0	104	92.1	24	54.5
vortex	2,161	3,573.9	43	92.5	14	81.0
wave5	863	579.3	51	94.6	20	84.7

Table 1: Summary of the SPEC95 benchmarks. The FORTRAN benchmarks are listed on top while the C benchmarks are listed below. See text for further explanation.

for a benchmark with complex control flow such as *gcc*. This represents a huge amount of execution frequency that can be assigned definitively (via the enumeration algorithm) to a fixed set of paths.

Figure 12(b) presents the ratio of the total flow to the potential flow integral for each benchmark, using the ordering of benchmarks from Figure 12(a). The potential flow is in the denominator since it will always be greater than or equal to the total flow. At first, one might expect this ratio to be close to one when the definite ratio is also close to one. However, if a benchmark executes a large number of paths there may be exponentially many paths with non-zero potential frequency. As a result, the potential integral can be much large than the total flow even when the definite integral is close to the total flow. Benchmarks such as *hydro2d*, and *gcc* exhibit such behavior.

4.3 Edge Profiles vs. Path Profiles

We now measure how well three different hot path predictors (based on edge profiles) perform compared to an actual path profile. The three predictors are:

- Definite predictor: enumeration of definitely hot paths in decreasing order of frequency (that is, the enumeration algorithm of Figure 11 applied to map $M_{\hat{D}}$).
- Potential predictor: enumeration of potentially hot paths in decreasing order of frequency.
- Greedy predictor: this predictor first marks all edges "unvisited"; as long as there is an unvisited edge, the algorithm picks an unvisited edge e of maximal frequency, and constructs a path (from *entry* to *exit*) by going forward from the $tgt(e)$ and backward from $src(e)$ in the DAG, always choosing an outgoing (incoming) edge with maximal frequency, regardless of whether or not it is marked "visited"; all edges in the newly generated path are then marked "visited". The order in which paths are generated reflects their hotness.

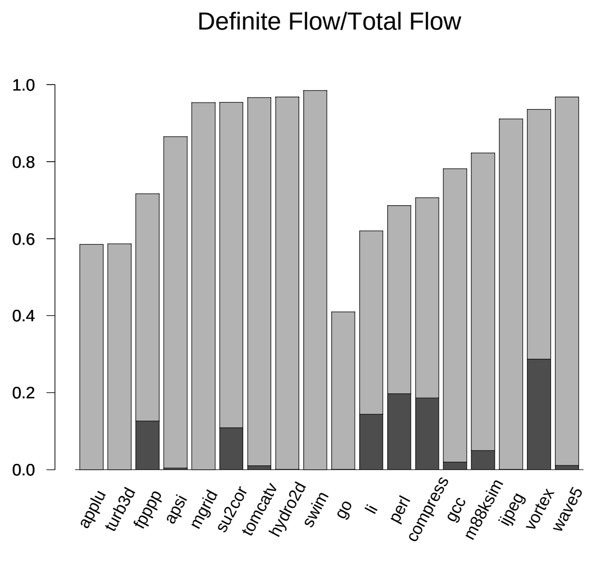
In order to examine how good the algorithms are at picking the hot paths globally over an entire benchmark, we construct a supergraph for each benchmark from the set of DAGs representing the procedures in the benchmark. The supergraph for benchmark B simply connects a unique entry vertex $entry_B$ for the benchmark to the entry vertices of all DAGs in the benchmark, and connects the exit vertices of all DAGs to a unique exit vertex $exit_B$. The edges $entry_B \rightarrow entry_P$ and $exit_P \rightarrow exit_B$ are assigned the frequency $freq(entry_P)$, where $entry_P$ ($exit_P$) is the entry (exit) vertex of procedure P's DAG. The edge frequencies for all other edges have already been determined by the edge profile for each procedure. As the supergraph is clearly a DAG, we can run the $M_{\hat{D}}$ and $M_{\hat{P}}$ algorithms on this graph and use the three predictors to pick hot paths over the entire benchmark, rather than on a per-procedure basis.

We use Wall's weight matching scheme [Wal91] to compare the paths picked by the three predictors to paths in the actual path profile. First, the executed paths (over all procedures) are totally ordered by their frequency in the actual path profile. A hot path in the actual profile is one whose frequency is greater than a certain percentage q of the total flow. A particular threshold q will select a set A_q of hot paths in the actual profile. The flow represented by these paths is

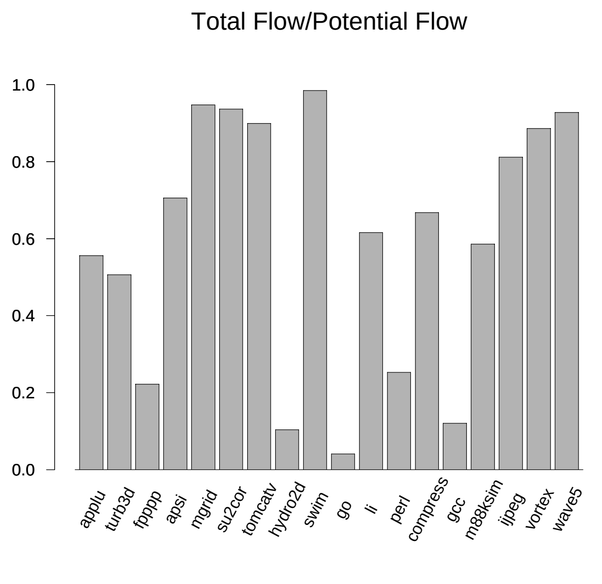
$$\sum_{p \in A_q} pfreq(p)$$

We then run each of the three predictors to get the top $|A_q|$ paths for that algorithm.[4] Let the paths generated by the

[4]The definite and greedy predictors may not be able to pick $|A_q|$ hot paths, as the definite enumeration algorithm can enumerate at most $width(G)$ paths, while the greedy algorithm can enumerate at most $|E|$ paths. Note that the potential algorithm can enumerate as many paths as in the graph.

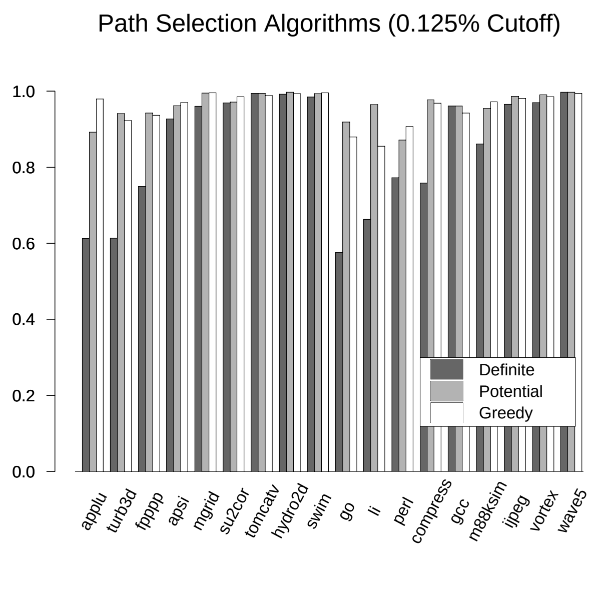


(a)

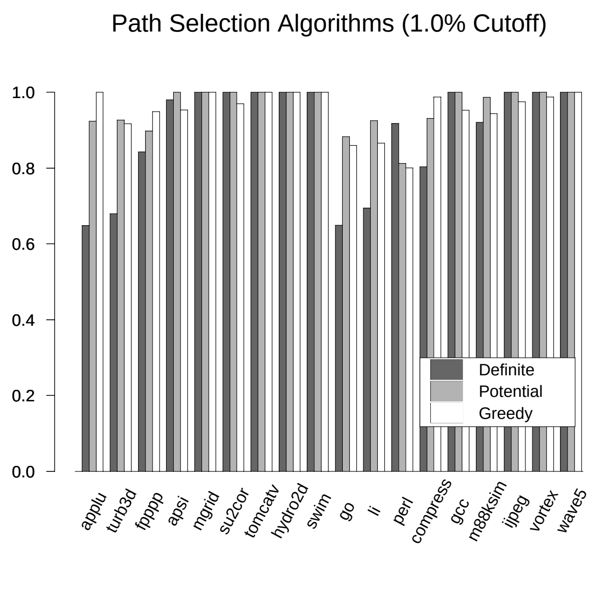


(b)

Figure 12: The ratio of definite flow to total flow in the SPEC95 benchmarks (a). The grey bars show the ratio for the total amount of definite flow in a benchmark, while the black bars show the ratio for the definite flow accounted for by procedures that executed exactly one path. The bar chart on the right (b) shows the ratio of the total flow to the potential flow integral, using the same ordering of benchmarks as in (a).



(a)



(b)

Figure 13: Results of weight matching the three hot path predictors (definite, potential, and greedy) for q cutoffs of 0.125% (a) and 1.0%. The y-axis represents the ratio R_Q (see text for explanation). A ratio close to 1.0 means that the predictor is picking the same hot paths as in the actual path profile.

definite, potential and greedy predictors be denoted by D_q, P_q, and G_q, respectively. Then the score for a particular set of paths Q with respect to A_q is the ratio:

$$R_Q = \sum_{p \in (A_q \cap Q)} \mathit{pfreq}(p) \;/\; \sum_{p \in A_q} \mathit{pfreq}(p)$$

This weighted ratio represents how well a particular predictor does at picking the paths in A_q. A predictor is penalized (indirectly) by picking paths outside of A_q.

We ran this experiment for values of q set to 0.125% and 1.0%, the same values used in Table 1. Figure 13 presents the results for the ratios R_{D_q} (Definite), R_{P_q} (Potential), and R_{G_q} (Greedy). The benchmarks are presented in the same order as in Figure 12. Both plots show that all three algorithms perform very well over all benchmarks. As expected, the definite predictor is handicapped by its inability to pick more than a sub-linear number of paths (note the similarity between the shape of the plot for the definite predictor and the shape of the plot in Figure 12(a)). The potential and greedy predictors perform very well: over both cutoffs, they typically exceed a 90% hit rate and only once dip slightly below 80%. The potential and greedy predictors have poorer performance when there is less definite flow in a benchmark, although they still perform remarkably well for some benchmarks with low definite flow (such as *applu*, *turb3d*, *go* and *li*). In general, we have found that the larger the q cutoff, the better the predictors do, which is very encouraging: they are picking the top hot paths very accurately.

5 Related Work

There is a very strong connection between our work and algorithms for linear programming and network flow, which we address first.

5.1 Linear Programming

By formulating flow equations that relate edge frequencies to path frequencies, one can find exact bounds for the frequency of a single path or a set of simultaneous paths using linear programming, as described below. In some sense this is more general than our result, which find bounds for single paths rather than sets of paths. Unfortunately, linear programming requires that the set of paths be explicitly encoded in the equations. Since the number of paths is exponential in the size of the DAG, the resulting system of equations can be enormous. However, once we have selected a subset of hot paths using our algorithms, linear programming can be used to get achieve better lower and upper bounds for sets of selected paths because it is able to find bounds on the flow achievable simultaneously by a set of paths.

Consider a set of acyclic paths P. Each path $p \in P$ has a variable x_p, which represents the frequency of p. We have already seen that given a set of acyclic paths P, each edge generates a constraint equation of the form $\mathit{freq}(e) = \sum_{p \in P(e)} x_p$. The set of edge equations can expressed in the form $A \cdot x = b$, where: A is a matrix of 0s and 1s coefficients in which each column represents an x_p and each row represents an edge's equation; x is a vector representing the path variables; and b is a vector representing the edge frequencies in an edge profile. The definite frequency for a path x_p is found by minimizing the objective function $c \cdot x$ subject to $A \cdot x = b$, where c is a vector containing a 1 at the entry corresponding to x_p and 0 everywhere else. This corresponds exactly to our computation of definite frequencies. Maximizing the objective function corresponds to finding potential frequencies. By setting multiple entries of the c vector to 1, one can find lower/upper bounds on the total flow achievable simultaneously by a set of paths.

5.2 Network Flow

The interpretation of the the control flow graphs under consideration here is clearly similar to that underlying network flow optimization. There are, however, some important differences in assumptions and objectives.

In traditional network flow problems the edge weights are interpreted as flow bounds (capacities), and the objective is to maximize overall flow (or to find a feasible flow, in the case where lower bounds are applied). In our problem, on the other hand, it is assumed that the weights have been generated by a feasible flow, and satisfy flow conservation at the vertices. The maximum flow is known, and the objective is to obtain information about flows along individual paths. Path-constrained maximum flow [GJ79], in which optimization applies to a subset of the paths of a graph, could be applied to each path, but this is unlikely to be efficient; nor does it take advantage of flow conservation at vertices.

The problems considered here are most similar to the closely related path optimization problems, where the edge weights are often interpreted not as flow bounds, but as lengths or costs. The single-source bottleneck path problem is to find the path from source to sink that maximizes the capacity of the lowest-capacity edge in the path (or minimizes the length of the longest edge in the path). This problem can be solved by suitably modifying a shortest path algorithm [EK72]. The potential frequency problem is a generalization of the bottleneck problem, where information about multiple paths is added. The algorithm itself can be thought of as a generalization of the bottleneck algorithm for an acyclic graph.

The definite frequency problem can be posed with an interpretation of the edge weights as flow lower bounds. However, since it makes a universal rather than an existential claim, there is no obviously related path optimization problem. Moreover, for the algorithm given here to be correct, it is important that the edge weights represent a feasible flow rather than arbitrary flow bounds. Though it may be possible to generalize the algorithm, we do not pursue this case here.

5.3 Other Related Work

Ball and Larus [BL96] compare paths chosen by path profiling with those chosen by a greedy algorithm using an edge profile [CMH91]. We take this work a step further by showing how to determine when an edge profile is a good predictor for hot paths and when it is not, which can indicate when the greedy algorithm or other hot path selection algorithms (such as the definitely and potentially hot path enumeration) will have good accuracy

Recent work by Tebaldi and West has addressed how to use Bayesian inference techniques to estimate path counts from link count data [TW96]. They use a linear algebraic formulation of the problem identical to that for linear programming (see above). However, the similarity ends there: they do not make use of lower or upper bounds in their work and require very expensive statistical computations.

Other related work shows to estimate vertex profiles via static analysis of the control flow graph [Wal91, WMGH94]. In contrast, we are estimating path profiles from existing edge profiles. Our results might be improved by performing static analysis as well. We could also apply our techniques to the vertex profiles estimated via static analysis, or collected by other techniques (such as PC sampling [ABD$^+$97]), in order to find hot paths.

6 Conclusions

We have provided a theoretical and algorithmic basis for characterizing the power of edge profiles to predict hot paths. This is captured in a simple dynamic programming algorithm that computes the amount of definite or potential flow in a graph, and an algorithm for enumerating definitely and potentially hot paths. If a program has a large amount of definite flow relative to its total flow, then hot path predictors such as the potentially hot path predictor and even greedy predictors will perform very well, as shown by our experimental results. On the other hand, if there is a small amount of definite flow, the behavior of such predictors is much more variable and path profiling may be needed to determine the hot paths precisely.

For the SPEC95 benchmarks, edge profiles are excellent predictors of hot paths, even for programs with complex conditional control flow. Further experimentation with other benchmarks is clearly needed to determine if our results generalize to other classes of programs.

Acknowledgments

Thanks to Audris Mockus and Ken Cox for many helpful discussions. Thanks also to Mark Staskauskas, Satish Chandra, and Jim Larus for their comments.

References

[ABD$^+$97] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S-T. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operaing Systems Principles*, October 1997.

[ABL97] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, June 1997. Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation.

[BL93] T. Ball and J. R. Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–13, June 1993. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.

[BL94] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[BL96] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, December 1996.

[CMH91] P. P. Chang, S. A. Mahlke, and W-M. W. Hwu. Using profile information to assist classic code optimizations. *Software–Practice and Experience*, 21(12):1301–1321, December 1991.

[CmWH88] P. Chang and Wen mei W. Hwu. Trace selection for compiling large c application programs to microcode. In *Proceedings of the 21st Annual WOrkshop on Microprogramming and Microarchitectures*, pages 21–29, San Diego, CA, November 1988.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, N.J., 1963.

[EK72] K. Edmonds and R. M. Karp. Theoretical improvements in algorthmic efficiency for network flow problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

[FF92] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. *ACM SIGPLAN Notices*, 27(9):85–95, October 1992. Proceedings of the 5th International Conference on Architectural Support for Programmming Languages and Operating Systems.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.

[GKM83] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software–Practice and Experience*, 13:671–685, 1983.

[KS73] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13:313–322, 1973.

[LS84] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6 – 22, January 1984.

[PSR92] S-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. *ACM SIGPLAN Notices*, 27(9):76–84, September 1992. Proceedings of the 5th International Conference on Architectural Support for Programmming Languages and Operating Systems.

[Smi81] J. E. Smith. A study of branch prediction strategies. *SIGARCH Newsletter*, 9(3):135–148, May 1981. Proceedings of the 4th Annual International Symposium on Computer Architecture.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for industrial and applied mathematics, Philadelphia, PA, 1983.

[TW96] Claudia Tebaldi and Mike West. Bayesian inference on network traffic using link count data. Technical report, Institute of Statistics and Decision Sciences Working Paper 16, Duke University, 1996.

[Wal91] D. W. Wall. Predicting program behavior using real or estimated profiles. *ACM SIGPLAN Notices*, 26(6):59–70, June 1991. Proceedings of the 1991 ACM Conference on Programming Language Design and Implementation.

[WMGH94] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. *ACM SIGPLAN Notices*, 29(6), June 1994. Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation.

A Proof of Theorem 1: Definite Paths

Theorem 1 *A path $p \in P(G_v)$, $v \in V$ has non-zero definite frequency f if and only if*

$$f \leq B_D(p) = F - \sum_{e \in p} \sum_{e' \in join(e)} freq(e')$$

Proof: Fix v and p throughout this proof. The theorem will hold if it can be shown that $\max\{B_D(p), 0\}$ is an exact lower bound on the frequency of p over all possible admissible frequency assignments. For an admissible frequency assignment *pfreq*,

$$F = \sum_{q \in P(G)} pfreq(q)$$

and

$$freq(e') = \sum_{q \in P(e)} pfreq(q)$$

so that

$$\begin{aligned} B_D(p) &= \sum_{q \in P(G)} pfreq(q) - \sum_{e \in p} \sum_{e' \in join(e)} \sum_{\substack{q \in P(G) \\ e' \in q}} pfreq(q) \\ &= \sum_{q \in P(G)} pfreq(q)\,(1 - J(p,q)) \end{aligned}$$

where

$$J(p,q) = \left|\{(e,e') \mid e \in p, e' \in q, e' \in join(e)\}\right|$$

is the number of times that q joins p. Since terms for which $J(p,q) = 1$ disappear from the above equation, we can rewrite it as:

$$B_D(p) = \sum_{\substack{q \in P(G) \\ J(p,q)=0}} pfreq(q) - \sum_{\substack{q \in P(G) \\ J(p,q)>1}} pfreq(q)\,(J(p,q)-1)$$

For $p \in P(G_v)$ and $q \in P(G)$, $J(p,q)$ is zero only when p is a suffix subpath of q. The frequency of the path p is the sum of the assigned frequencies of all paths in $P(G)$ of which it is a suffix:

$$\begin{aligned} subpfreq(p) &= \sum_{\substack{q \in P(G) \\ J(p,q)=0}} pfreq(q) \\ &= B_D(p) + \sum_{\substack{q \in P(G) \\ J(p,q)>1}} pfreq(q)\,(J(p,q)-1) \quad (1) \end{aligned}$$

This expression may be used to prove the two lemmas below, which together show that $\max\{B_D(p), 0\}$ is indeed an exact lower bound.

Lemma A.1 *For any admissible frequency assignment pfreq,*

$$subpfreq(p) \geq \max\{B_D(p), 0\}$$

□

Proof: Trivial from expression (1).

Lemma A.2 *For any flow-conserving edge profile freq, an admissible frequency assignment pfreq exists for which*

$$subpfreq(p) = \max\{B_D(p), 0\}$$

□

Proof: It immediately follows from expression (1) that

$$subpfreq(p) = B_D(p) \iff \forall q \in P(G), J(p,q) > 1 \Rightarrow pfreq(q) = 0$$

To prove the lemma, it suffices to show that some *pfreq* exists that satisfies either

$$\forall q \in P(G), J(p,q) > 1 \Rightarrow pfreq(q) = 0$$

or

$$subpfreq(p) = 0$$

This may be demonstrated by the following iterative construction. Let the assigned frequencies at any stage of the construction be given by $\overline{pfreq}(q), q \in P(G)$, and define residual edge frequencies

$$\overline{freq}(e) = freq(e) - \sum_{q \in P(e)} \overline{pfreq}(q)$$

Note that the residual edge frequencies satisfy flow conservation. This means that for any edge with non-zero residual frequency, a path exists from *entry* to *exit* containing that edge, consisting only of edges with non-zero residual frequency. Initialize the assigned frequencies ($\overline{pfreq}$) to zero, then apply the iterative procedure **Reduce**:

(Reduce)

1. If any edge in p has zero residual frequency, stop.
2. If all edges joining p have zero residual frequency, stop.
3. Pick an edge $\bar{e}$ joining p for which $\overline{freq}(\bar{e}) > 0$, and construct a path $\bar{q} \in P(G)$ that joins p exactly once, by finding
   - a suffix subpath of p that begins at $tgt(\bar{e})$ (and ends at *exit*)
   - a subpath from *entry* to $src(\bar{e})$ that doesn't join p (though it may include a prefix subpath of p)

   Such subpaths can always be found, because of flow conservation and availability of p edges.
4. Increment $\overline{pfreq}(\bar{q})$ by one, and repeat.

Now apply procedure **Complete** to the residual configuration:

(**Complete**)

1. If all edges have zero residual frequency, stop.
2. Pick any path $\bar{q} \in P(G)$ such that all the edges in $\bar{q}$ have non-zero residual frequency.
3. Increment $\overline{pfreq}(\bar{q})$ by one, and repeat.

Reduce assigns frequencies only to paths $\bar{q}$ such that $J(p, \bar{q}) = 1$. If **Reduce** terminates at step 1, **Complete** assigns frequencies only to paths such that $J(p, \bar{q}) > 0$, since no path containing p can ever be chosen; in this case, no path containing p ever has its frequency incremented, so $subpfreq(p) = 0$. Conversely, if **Reduce** terminates at step 2, **Complete** assigns frequencies only to paths such that $J(p, \bar{q}) = 0$. In this case, no path joining p more than once ever has its frequency incremented. Since **Reduce** must terminate in one of these two ways, the frequency assignment constructed satisfies at least one of the requisite properties, and the lemma is established.

The result stated in the theorem immediately follows. □

B Proof of Theorem 2: Number of Definite Paths with Non-zero Frequency

For a given edge profile, let the set of paths from vertex v to *exit* having non-zero definite frequencies be $P_{def}(G_v) \subseteq P(G_v)$. Then Theorem 2 may be restated as

Theorem 2

$$|P_{def}(G_v)| \leq width(G_v)$$

Proof: First observe that the paths in $P_{def}(G_v)$ have some special structure:

Lemma B.1 *Any two distinct paths $p_1, p_2 \in P_{def}(G_v)$ satisfy $J(p_1, p_2) = 1$.* □

Proof: Since p_1 and p_2 both start at v and end at *exit*, but are distinct, $J(p_1, p_2) \neq 0$. Assume that $J(p_1, p_2) > 1$. Since p_1 has a non-zero definite frequency, a frequency assignment exists under which all paths $q \in P(G)$ containing p_2 have zero frequency, since $J(p_1, q) > 1$ for all such paths. This contradicts the assumption that $p_2 \in P_{def}(G_v)$. □

This result allows the following lemma to be established, from which the theorem follows. Let G_{def} be the directed subgraph of G_v defined by edges contained in paths in $P_{def}(G_v)$.

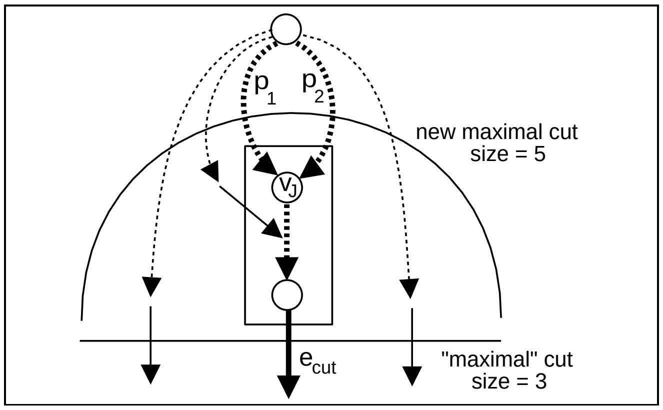


Figure 14: Possible configurations for paths p_1 and p_2 sharing an edge $e_{\rm cut}$ in $E_{\rm cut}$. Dotted lines represent paths while solid lines represent edges. No edges other than $e_{\rm cut}$ can leave the subpath from v_J to the source of edge $e_{\rm cut}$, delineated by the box.

Lemma B.2 *Given any maximal cut of G_{def}, let the set of edges crossing the cut be $E_{\rm cut}$. Then*

$$|E_{\rm cut}| = |P_{def}(G_v)|$$

□

Proof: By definition of the cut, each path in $P_{def}(G_v)$ has exactly one edge in $E_{\rm cut}$, and each edge in $E_{\rm cut}$ is a member of at least one path in $P_{def}(G_v)$. The lemma is established if no edge in $E_{\rm cut}$ is a member of more than one path. Assume that an edge $e_{\rm cut} \in E_{\rm cut}$ is shared by two paths $p_1, p_2 \in P_{def}(G_v)$. From the result above, p_1 and p_2 join exactly once. Since they share $e_{\rm cut}$, either they join at some vertex v_J that occurs no later than $src(e_{\rm cut})$ in their common subpath, or they diverge at some vertex v_D that occurs no earlier than $tgt(e_{\rm cut})$.

Consider the former case (see Figure 14): no third path in $P_{def}(G_v)$ can diverge from the common subpath at any vertex between v_J and $src(e_{\rm cut})$, inclusive, because such a path would join at least one of p_1, p_2 in two places. But this means the cut is not maximal, since it can be modified to pass on the other side of v_J, increasing the number of edges in the cut set by at least one (more than one if there are joining edges at vertices between v_J and $src(e_{\rm cut})$). Likewise, in the latter case (not shown), no third path can join at any vertex between $tgt(e_{\rm cut})$ and v_D, inclusive. Again, the cut is not maximal, since it can be modified to pass on the other side of v_D, increasing the number of edges in the cut set. Thus the original assumption is incorrect, and no edge is a member of more than one path. □

Since the width of the reduced graph is, by definition, the cardinality of a maximal cut edge set,

$$|P_{def}(G_v)| = width(G_{def}) \leq width(G_v). \text{ □}$$

Action Transformation:

An Application of Sort Inference

by Kent D. Lee

University of Iowa

Advisor: Ken Slonneger

Overview

- Introduction to Action Semantics
- Automated Compiler Generation
- Example Calculator Language
- Action Transformations

Action Semantics

- Created by Peter Mosses with collaboration of David Watt

- Defines the action as the basic unit of computation

What's the Purpose of an action?

- Actions manipulate three structures
  - Transients - Represented as tuples of data
  - Bindings - Represented as a map of identifiers to bindable data
  - Storage - Represented as a map of cells to storable data
- Actions compute new transients, bindings, and storage from existing transients, bindings, and storage.
  - Incoming transients, bindings, and storage
  - Outgoing transients, bindings, and storage

Example Actions

- allocate a cell
  - sets aside a new cell in storage and gives it as a transient
- bind “M” to 5
  - produces the new binding in the outgoing bindings
- store 6 in the given cell
  - stores 6 by mapping the given cell to the value 6.

Yielders

- Sometimes we want to access the incoming transients, bindings, or storage.

- bind “M” to the **given cell**
  - **given** s is a yielder that accesses the first datum in the incoming transients and insists that it is a subsort of sort s

- **given** $s\#n$ accesses the n^{th} datum of the incoming transients

- s **bound to** id accesses the datum that id maps to in the incoming bindings

- s **stored in** c accesses the datum mapped to by the cell c

Data

- Action Semantics uses a sort system instead of a type system.
- Sorts of data are partially ordered by the subsort relation ($\leq$)
- datum is a sort that contains all individuals
- Other sorts include integers, truth-values, reals, 4, 5, true|false (i.e. truth-value), integer&truth-value, nothing, and data
- $s_1|s_2$ is the sort of the join of s_1 and s_2
- $s_1\&s_2$ is the sort of the meet of s_1 and s_2

Combinators

- Simple Actions can be combined to create complex or compound actions

- Combinators serve two purposes
  - They are the glue for tying actions together
  - They dictate how the incoming and outgoing transients, bindings, and storage of the subactions will be combined in the composite action

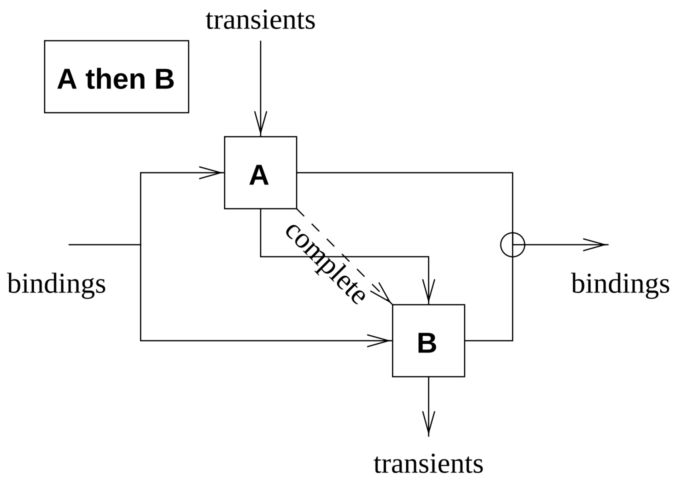
Combinator Example

allocate a cell **then** bind “M” to the given cell

- **then** is a combinator that passes the outgoing transients of the first sub-action to the incoming transients of the second sub-action
- There are many ways of propogating transients, bindings, and storage among the subactions of an action.
- If we restrict ourselves to deterministic actions, then storage is single threaded and can be treated as a global store
- To understand combinators we can look at how the transients and bindings are propogated

Combinator Diagrams

Devised by Ken Slonneger, they help us visualize how transients and bindings are propogated between the sub-actions of a combinator



Execution proceeds from the upper left to the lower right

Overview Notes

Slow Down.

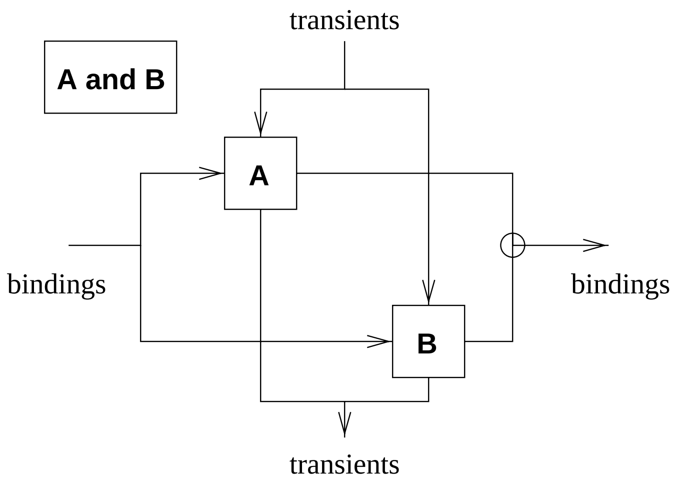
In this picture sub-action A receives the transients of the combined action. A's outgoing transients become B's incoming transients.

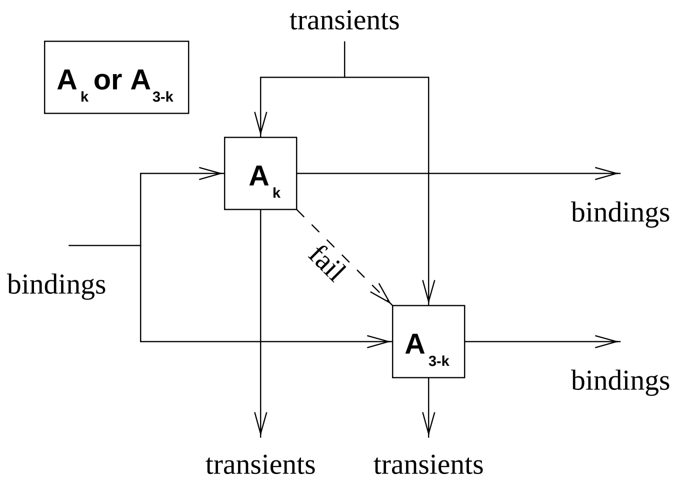
B's outgoing transients are the outgoing transients of the combined action

The bindings passed to the combined action are passed on to the subactions.

The bindings produced by A and B are merged to produce the bindings of the combined action.

Some Other Combinators





Action Semantic Definitions

- Like other semantic formalisms we wish to construct a mapping from the concrete syntax of a language to its semantics (actions).

- My implementation skips the abstract syntax representation

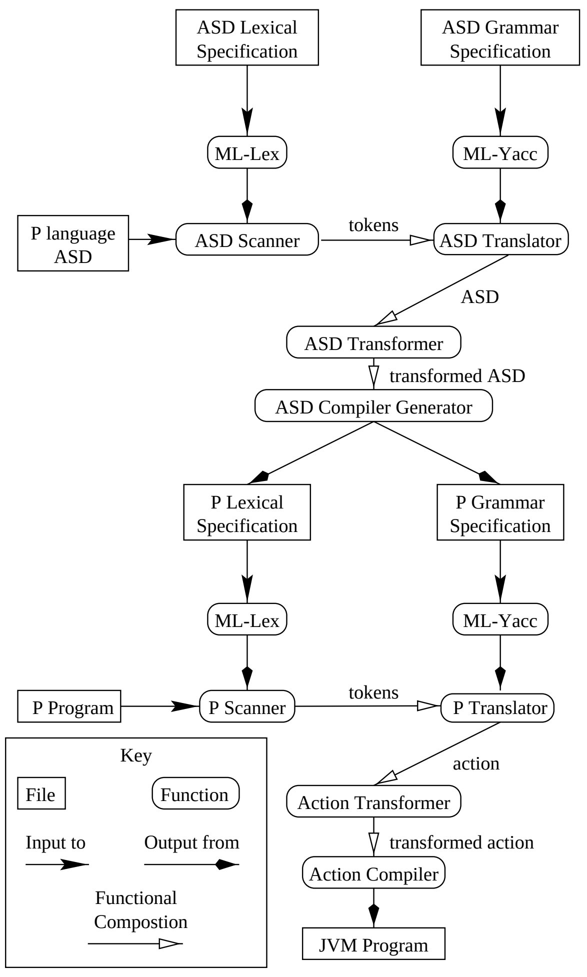
- Action Semantic Descriptions are modular.

- They are easier to extend than say a denotational description

Put up the calculator asd

Automated Compiler Generation

Using an Action Semantic Description (ASD) it's straightforward to write a parser generator that given an ASD and a source program will map the program to it's action



Calculator Program

- For example, consider the program 4+5=
- It can be mapped to its action using the ASD above.
- Its action is:

Put up the slide on the 4+5= program

Problem

- We want to generate efficient code from the program action
- Actions require bindings and transients to be passed from action to action
- Passing transients and bindings at run-time would be costly
- Traditional compilers for statically bound languages don't pass transients or bindings at run-time
- How should we eliminate transients and bindings from actions of statically bound languages?

Two Approaches

1. We could make the code generator very smart and have it work at eliminating the transients and bindings. This approach was used by Peter Orbaek in his compiler generator, Oasis.

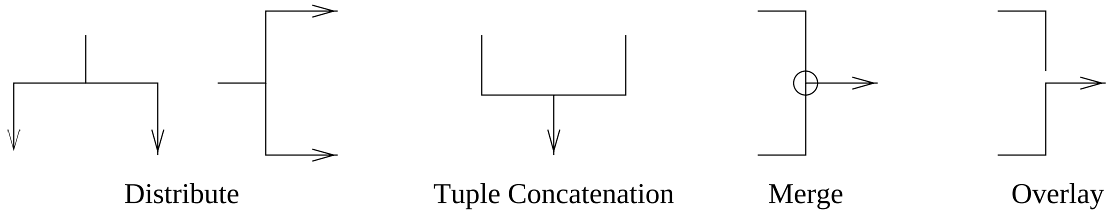
2. Or we could take the approach put forward by Watt, Brown, and Moura. They perform sort inference on the program's action and use the information obtained to eliminate the transients and bindings.

Action Sort Inference

- Based on Record Type Inference described by David Schmidt and Susan Even.
- Deryck Brown extended it to work in his Actress Compiler Generator
- Records are maps from identifiers to sorts
- Four records, one for each of the incoming and outgoing transients and bindings, describe the sort of an action.
- Storage is ignored since it's domain can't be known statically
- Brown defines five operators on Records that together implement sort inference over all action combinators

Record Sort Operations

- Distribute, Merge, Switch, Select, and Overlay are the operators Brown proposes. These operators correspond to the operations shown in the combinator diagrams above.



- The Actress Compiler Generator implements a dialect of Action Semantics that forces you to give transients in an odd way

- The Tuple Concatenation operator enables my compiler generator to represent Actions as described by Mosses in his text on the subject.

Sort Inference Example

- A complete example would be too tedious to cover here
- It should be sufficient to say this is one of those “left as an exercise for the reader” problems.
- After applying sort inference to our example above (i.e. 4+5=) we get the annotated action found here

Put up test0.sorts

Action Transformations

- By applying sort inference carefully it is possible to use the sort information to remove all transients and bindings from statically bound languages.

- In the example θ_0, θ_4, and θ_5 appear where ever the cell bound to “M”, 4, or 5 appear respectively

- Moura describes these transformations in his thesis

Transient Elimination

- Transients can be eliminated by using sort information to rewrite yielders that give values based on the incoming transients.

- For instance, the action

```
(give 4 and then give 5)

then

give sum(given integer#1,given integer#2)
```

- Can be rewritten as

```
give sum(4,5)
```

Transient Elimination Algorithm

- Transient Elimination is achieved by replacing all give actions with the null action - complete

- Replace all given yielders with their actual values

  (complete and then complete)

  then

  given sum(4,5)

- Utilize algebraic identities of actions to simplify

  complete and then $a_2 = a_2$

  complete then $a_2 = a_2$

Binding Elimination

- Binding elimination is performed the same way transient elimination is performed

- Bind actions are replaced with complete

- Bound to yielders are replaced with their actual values

- Algebraic Identities involving complete are used to reduce the action

Conclusion

- Once Transient and Binding elimination have been performed, code generation is relatively easy.

- In fact, the actions that result from applying transient and binding elimination look a lot like assembly language code.

- Future work includes applying sort inference to Action Semantic Descriptions to validate and simplify *descriptions* of programming languages in addition to applying sort inference to each individual program's action

```
||allocate [integer]cell
|then
||bind "M" to the (given [integer]cell) [[integer]cell]
hence
|||give the 4
||and then
|||give the 5
|then
||give the sum (given integer#1, given integer#2)
```

```
||allocate [integer]cell
||: Action ({},{}) -> ({1:theta_0},{})
|then
||bind "M" to the (given [integer]cell) [[integer]cell]
||: Action ({1:theta_0},{}) -> ({},{M:theta_0})
|: Action ({},{}) -> ({},{M:theta_0})
hence
|||give the 4
|||: Action ({},{M:theta_0}) -> ({1:theta_4},{})
||and then
|||give the 5
|||: Action ({},{M:theta_0}) -> ({1:theta_5},{})
||: Action ({},{M:theta_0}) -> ({1:theta_4,2:theta_5},{})
|then
||give the sum (given integer#1, given integer#2)
||: Action ({1:theta_4,2:theta_5},{M:theta_0}) -> ({1:theta_10},{})
|: Action ({},{M:theta_0}) -> ({1:theta_10},{})
: Action ({},{}) -> ({1:theta_10},{})
```

Action Transformation: An Application of Sort Inference

Kent D. Lee

May 30, 1998

Abstract

Action Semantics is a formal method of defining programming language semantics in which actions describe the manipulation of three entities: transients, bindings, and the store. Due to the high-level nature of Action Semantics, actions cannot be directly translated into efficient code in an Action Semantics-based compiler. However, by applying sort inference to an action, it is possible to transform it to a *low-level* action that can be translated into efficient code. This paper briefly introduces Action Semantics. The problem of Action Semantics-directed compilation is then demonstrated through an example. Finally, sort inference is applied to an action and it is subsequently transformed to an action more suitable for compilation.

1 Introduction

Action Semantics is a formal method of defining programming languages developed by Peter Mosses[9, 10] in collaboration with David Watt[15]. Action Semantics is based on Denotational Semantics. But Action Semantics, unlike Denotational Semantics, is designed to be both modular and readable by average programmers. A modular language specification is one that can be extended to introduce new language constructs with little or no rewriting of the existing description[10].

Another semantic method with similarities to Denotational Semantics is Monadic Semantics. Monadic Semantics exhibits modularity like Action Semantics. Monad Transformers allow the language designer to develop modular language descriptions that, through the use of partial evaluation, may be used to develop both modular interpreters and compilers[6, 7, 8]. However, Monadic descriptions are not suitable as language descriptions for the average programmer since an in depth knowledge of category theory is a prerequisite[10].

Compiler generation based on Action Semantics suffers from many of the same problems as compilers based on Monad Transformers. Actions, the computational entity of Action Semantics, are very high-level and not easily translated into a low-level form like assembly language. One approach to this problem is to make the code generator very complex and perform optimizations on the target program to simplify it. This is the approach taken in [13]. However, through the use of sort inference, an action may be simplified to a form that resembles assembly language[12, 11]. The same effect has been achieved through the use of partial evaluation[3, 4] and by the definition of a special low-level compilation semantics in the context of Monad Transformers[6].

This paper describes the problem of translating an action to a form suitable to be used in code generation in a compiler. In particular, this paper focuses on code generation for statically bound languages. Section 1.1 contains a brief introduction to Action Semantics. The reader familiar with Action Semantics can skim this section while paying careful attention to the description of combinators. For a more thorough introduction see[14]. Mosses has also written a definitive text on Action Semantics[10]. Section 2 presents an Action Semantic description and describes a compiler generator called Genesis based on Action Semantics. Section 3 describes the sort inference algorithm used in Genesis. Section 4 shows how the sort inference information may be used to eliminate transients and bindings from a program's action. Section 5 concludes and describes areas of future research in the Genesis project.

1.1 Action Semantics

In Action Semantics the action is the basic unit of computation. An action manipulates one or more of three entities: transients, bindings, and the store. Transients are intermediate results of computation. Bindings are the usual definition, a mapping of identifiers to values or locations within the store. The store is a mapping of locations, called cells, to values. In the language of Action Semantics, actions give transients, produce bindings, and store data.

Transients, bindings, and storage are all ways of organizing data. Action Semantics uses a sort system instead of a type system. Sorts are partially ordered by the subsort operator (i.e. $\leq$) with the special sort nothing as the least sort. In a sort system there is no distinction between an individual and the sort containing just that individual. For example, 6 and the sort of 6 are the same thing. Datum is the sort of all individual sorts. Data is another name for the sort of tuples of datum.

In this paper integers and cells are used, but other sorts are defined in Action Semantics and more can be added. Standard Action Semantics includes sorts such as truth-value, real, character, and map. Unlike standard Action Semantics, the Genesis compiler generator qualifies cells (a map from natural to storable) according to the datum they may contain. So instead of a cell, Genesis calls a generic cell a [datum]cell. A cell that must hold an integer is called an [integer]cell. This distinction is important to efficiently generate code for a given action.

Actions may give transients. For example,

    allocate a cell

is an action that dynamically allocates a cell and gives it as a transient. The action

    bind "M" to 5

is another action that produces a new binding of M to 5. Actions can also store values as in

    store 6 in the given cell

This action stores 6 in the cell that's given to the action. Actions not only give transients, produce bindings, and store values, but their performance can depend on the given transients, produced bindings, or stored values. In other words, an action's performance may depend on incoming transients, bindings, and storage. Every action creates some outgoing transients and bindings and may affect the contents of the store.

Actions that depend on the incoming transients, bindings, and storage contain yielders. The action

    store 6 in the given cell

contains the yielder given S which examines the incoming transients and yields the datum, d, contained in it such that $d \leq S$. To access the n^{th} element of the incoming transients the yielder given $S\#n$ may be used. Other yielders allow actions to access bindings and the store. The yielder S bound to id yields the d that's bound to id in the incoming bindings provided $d \leq S$. The yielder S stored in c yields d provided $d \leq S$ and d is stored in cell c.

Actions and yielders which use transients or give transients are said to be part of the *functional* facet of Action Semantics. Actions and yielders which produce or receive bindings are said to be part of the *declarative* facet of Action Semantics. Finally, actions and yielders which store or depend on the contents of the store are said to be part of the *imperative* facet of Action Semantics.

For actions to be interesting, we must be able to combine them into complex or compound actions. Combinators serve as constructors for complex actions. While storage is single threaded through actions (assuming no non-determinism, which isn't relevant in relation to compiler generation), transients and bindings may be propagated from one action to another in several different ways. For example, the action

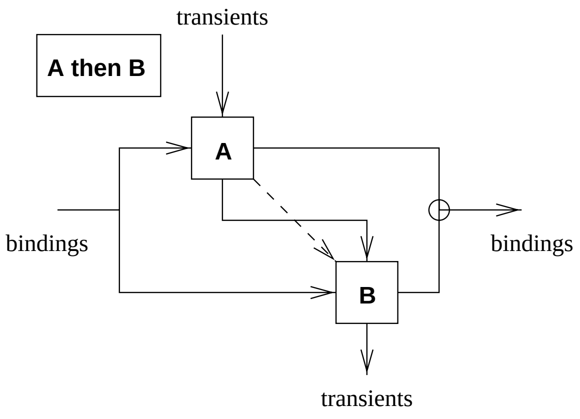


Figure 1: then combinator diagram

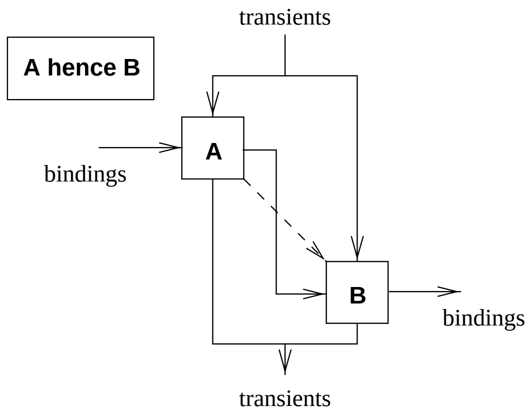


Figure 2: hence combinator diagram

```
| allocate a cell
then
| bind “M” to the given cell
```

contains the then combinator. A then B is an action that passes the outgoing transients of subaction A to the incoming transients of subaction B. The outgoing transients of B are the outgoing transients of the combined action. The incoming transients to A then B become the incoming transients to A.

To better understand combinators it is helpful to graphically see how different combinators propagate transients and bindings. Ken Slonneger [14] uses combinator diagrams to depict the nature of com-

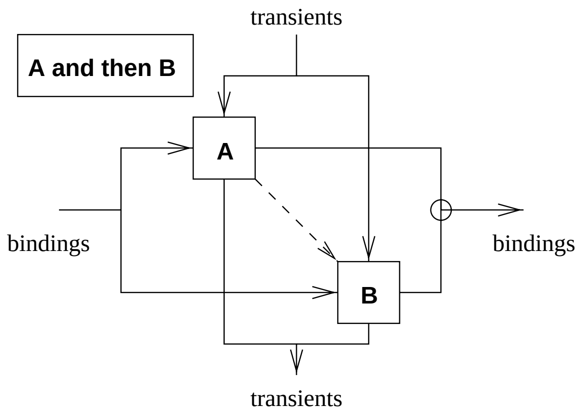


Figure 3: and then combinator diagram

binators. For instance, the then combinator is graphically depicted in figure 1. The bindings and transients are propagated as shown. The dashed line indicates that subaction A is performed before subaction B (i.e. stores performed in A will be visible to B). Some other combinators of interest are hence and and then depicted in Figures 2 and 3.

The action complete is the action that produces no bindings and gives no transients. It is of interest because it is a unit for and then and is used in algebraic identities involving the combinators then and hence.

2 A Semantic Description

To illustrate sort inference it's useful to have an example language. The calculator language will serve this purpose. The calculator is the language of expressions over integers with the addition of a memory location and the ability to store a value in it and recall its value. The language is defined via an Action Semantic description which is given in the appendix.

Action Semantic descriptions are analogous to Denotational descriptions. Semantic equations map syntax to semantics. However, the Genesis compiler generator disposes of abstract syntax and maps concrete syntactic phrases directly to their action phrases. The structure of Genesis is shown in figure 4. It generates a compiler that consists of a front end that compiles a source program to its action, and

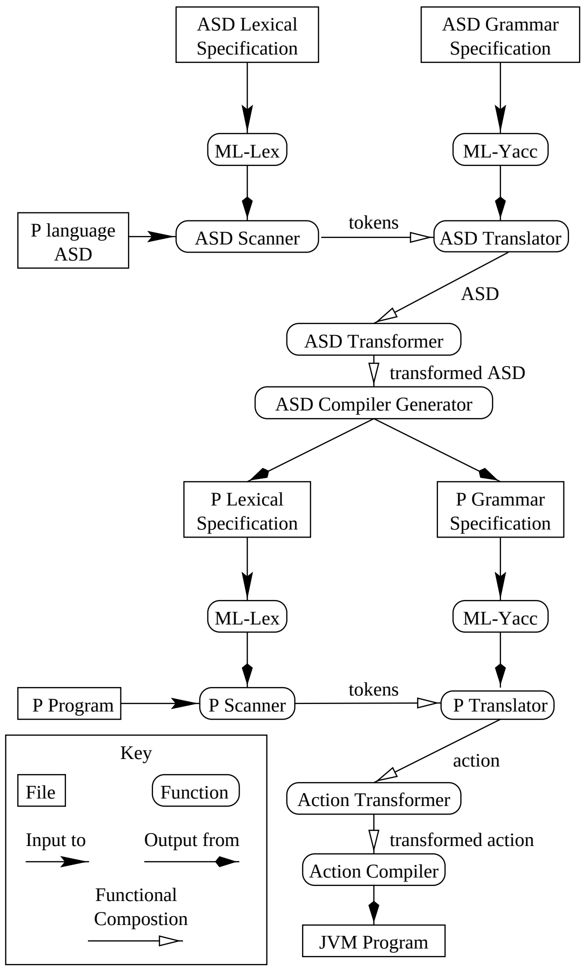


Figure 4: Genesis compiler generator

a back end that compiles an action to a target program. Initially the target program will be a Java assembly language program. However, the compiler generator can be retargeted easily by writing a back end for other target languages.

The front end generated by Genesis depends on the source language. It consists of a parser based on the concrete syntax derived from the Action Semantic description of the language. The parser translates a source program to its action. The back end is the same for every Genesis generated compiler. It consists of an Action Transformer that performs binding and transient elimination on the program action. The Action Transformer is responsible for the action transformations described in section 4.

The calculator semantics begins by allocating a cell and binding it to the identifier M. After transforming actions of the calculator language, the M binding will be eliminated and the residual action will consist of *low-level* actions that can be easily translated into assembly language. However, before transforming an action, it is necessary and useful to perform sort inference over the action. Sort inference over actions is the topic of the next section.

3 Sort Inference

Performing sort inference on actions means examining the incoming and outgoing transients and bindings to verify that they are consistent according with the meaning of the action combinators used in the action. An action's sort is the collection of these four sorts: the sorts of the incoming and outgoing transients and bindings for the action. The imperative facet is not considered in sort inference since the domain of the store cannot generally be known statically.

Each of the incoming and outgoing transients and bindings are described by a record. Transients are represented as a record that maps natural numbers to sorts. For bindings, the record maps identifiers to sorts. Sort inference on actions was first described by Even and Schmidt in [5] and was extended by Brown and Watt in [1, 2]. Watt, Brown, and Moura do not implement transients as a tuple in their compiler generator, called Actress. As a result they introduce a dialect of Action Semantics that treats transients as a binding of natural numbers to datum. The net result of this is that their dialect of Action Semantics is less modular than the original definition. This paper extends the sort inference algorithm of [5] and [2] to properly treat transients as tuples of datum.

Consider the action

| allocate a cell
then
| bind "M" to the given cell

After applying sort inference, this action has sort

$$(\{\},\{\}) \rightarrow\ (\{\},\{M : [\mathit{datum}]\mathit{cell}\})$$

The action takes no incoming transients or bindings and produces one binding.

Record sorts are extended to record sort schemes for the purpose of sort inference. Record sort schemes consist of a (possible) row variable and a collection of fields that map the record domain to its range. Field schemes may either be absent, present and represented as a sort scheme, or undetermined (represented as a field variable) to be resolved to present or absent later. Finally, sort schemes are either a subsort of datum or represented as a sort variable. For instance, assume the subaction

bind "M" to the given cell

is annotated with the sort scheme

$$(\{1 : \theta_1\}\gamma_3, \{\}\gamma_4) \rightarrow\ (\{\},\{\})$$

where the constraint $\theta_1 \& [\mathit{datum}]\mathit{cell} \neq \mathit{nothing}$ holds. γ_3 and γ_4 are row variables and θ_1 is a sort variable. Assume the action

allocate a cell

is annotated with the sort scheme

$$(\{\}\gamma_1, \{\}\gamma_2) \rightarrow\ (\{1 : [\mathit{datum}]\mathit{cell}\},\{\})$$

To find the sort of the entire action, the outgoing transients of the allocate action must be unified with the incoming transients of the bind action. The transients passed to the entire action (i.e. the empty transients) must be unified with the transients passed to the allocate action. The outgoing transients of the entire action are the same as the outgoing transients of the bind action.

The incoming bindings of the entire action (i.e. the empty bindings) must be unified with each of the incoming bindings of the subactions. The outgoing bindings of the two subactions must be merged together. All this information is contained in the fact that the then combinator is used to combine the two subactions.

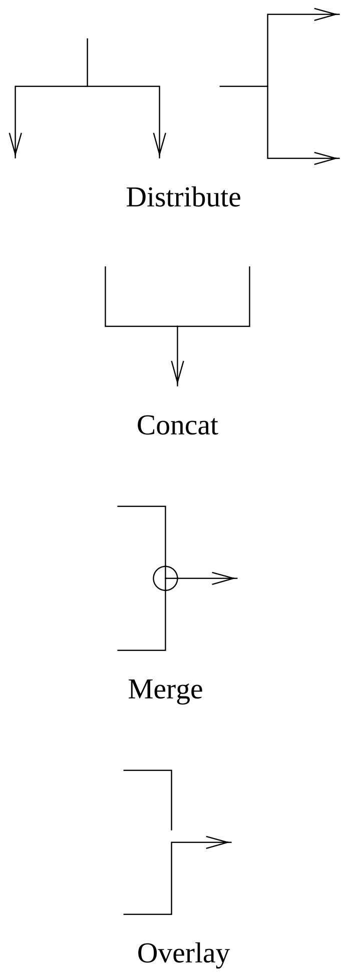


Figure 5: Derived Sort Inference Operators

Underlying the combinators used in Action Semantics are a set of derived sort inference operations which are used in the unification of record sort schemes. These derived operations are actually hinted at in the combinator diagrams of Slonneger [14]. Brown names some of these operations in [1]. The operations of interest in this paper are in figure 5. There are two other derived operations that are not described here that are needed in conjunction with the or combinator. The or combinator isn't used by the calculator language and so is omitted because of space constraints.

One derived operation is named in this paper that was avoided by Brown in [1] because of his treatment of transients. The *concat* operation is used to concatenate the outgoing transients of two subactions combined by the and then combinator. For instance, consider the actions

```
give 6
```

with sort scheme

$$(\{\}\gamma_1, \{\}\gamma_2) \to (\{1:6\}, \{\})$$

and

```
give 5
```

with sort scheme

$$(\{\}\gamma_3, \{\}\gamma_3) \to (\{1:5\}, \{\})$$

Combining these two actions with the and then combinator causes their outgoing transients to be concatenated as in

$$concat(\{1:6\}, \{1:5\}) = \{1:6, 2:5\}$$

The concatenation operator insists that the first record sort scheme be exactly known. So, $concat(\{1:6\}, \{1:5\})$ is defined whereas $concat(\{1:6\}\gamma_i, \{1:5\})$ would not be defined. This means that some polymorphic actions like

```
| give the given data
and then
| give 5
```

```
| | allocate [integer]cell
| | : $(\{\},\{\}) \to (\{1:\theta_{35}\},\{\})$
| then
| | bind "M" to the given cell
| | : $(\{1:\theta_{35}\},\{\}) \to (\{\},\{M:\theta_{35}\})$
| : $(\{\},\{\}) \to (\{\},\{M:\theta_{35}\})$
hence
| | | | | give the 6
| | | | | : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_6\},\{\})$
| | | | and then
| | | | | | give the 5
| | | | | | : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_7\},\{\})$
| | | | | then
| | | | | | | store the given integer in
| | | | | | | the cell bound to "M"
| | | | | | | : $(\{1:\theta_7\},\{M:\theta_{35}\}) \to (\{\},\{\})$
| | | | | | and then
| | | | | | | give the given integer
| | | | | | | : $(\{1:\theta_7\},\{M:\theta_{35}\}) \to (\{1:\theta_7\},\{\})$
| | | | | | : $(\{1:\theta_7\},\{M:\theta_{35}\}) \to (\{1:\theta_7\},\{\})$
| | | | | : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_7\},\{\})$
| | | | : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_6, 2:\theta_7\},\{\})$
| | | then
| | | | give the (sum (given integer#1,
| | | | given integer#2)) [ giving integer]
| | | | : $(\{1:\theta_6, 2:\theta_7\},\{M:\theta_{35}\}) \to (\{1:\theta_{20}\},\{\})$
| | | : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_{20}\},\{\})$
| | and then
| | | give the integer stored in the cell bound to "M"
| | | : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_{33}\},\{\})$
| | : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_{20}, 2:\theta_{33}\},\{\})$
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
| | : $(\{1:\theta_{20}, 2:\theta_{33}\},\{M:\theta_{35}\}) \to (\{1:\theta_{32}\},\{\})$
| : $(\{\},\{M:\theta_{35}\}) \to (\{1:\theta_{32}\},\{\})$
: $(\{\},\{\}) \to (\{1:\theta_{32}\},\{\})$
```

Figure 6: Calculator action for $(6 + 5S) * R =$

result in sort inference failure in the Genesis compiler generator. However, this type of action doesn't arise in practice and so the restriction of the derived concat operation doesn't seem to pose a problem.

Unifying record sort schemes is complex and is covered in detail in [1]. However, combining record sort schemes $A = \{fields_A\}row_A$ and $B = \{fields_B\}row_B$ can be briefly summarized as follows

1. Let $fields_{A'} = fields_B - fields_A$ and $fields_{B'} = fields_A - fields_B$.

2. Extend A and B into two new record sort schemes $A' = \{fields_A @ fields_{A'}\}row_{A'}$ and $B' = \{fields_B @ fields_{B'}\}row_{B'}$ whose field domains are identical. Instantiate the row variables of A and B to the newly created records (i.e. $[row_A \mapsto \{fields_{A'}\}row_{A'}, row_B \mapsto \{fields_{B'}\}row_{B'}]$)

3. Unify A' and B' according to the specified derived sort inference operation (i.e. Distribute, Concat, Merge, Overlay, Switch, or Select).

Sort inference in Genesis, like that of the Actress Compiler Generator[2], is carried out by the algorithm described above. At the end of sort inference, action sort schemes are reduced to a normalized form. A sort scheme is normalized if it is either a sort variable bound to a sort (not a sort scheme!) or a sort variable bound to a sort constructor applied to a normalized sort scheme. Normalization results in an action that is annotated with sort variables. For instance, after sort inference was performed, the calculator expression $(6 + 5S) * R =$ results in the annotated action found in figure 6 with the sort variable substitution

$$[\theta_6 \mapsto 6, \theta_7 \mapsto 5, \theta_{20} \mapsto integer,$$
$$\theta_{32} \mapsto integer, \theta_{33} \mapsto integer, \theta_{35} \mapsto [\theta_{33}]cell]$$

As might be expected the annotated action in figure 6 and the substitution given above contain valuable information that will be used in the next section.

```
| | allocate [integer]cell
| then
| | bind "M" to the given cell
hence
| | | | | complete
| | | | and then
| | | | | | complete
| | | | | then
| | | | | | | store 5 in the cell bound to "M"
| | | | | | and then
| | | | | | | complete
| | | then
| | | | give the (sum (6, 5)) [ giving integer]
| | and then
| | | give the integer stored in
| | | the cell bound to "M"
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
```

Figure 7: After eliminating θ_6 and θ_7

4 Transformation

While the action presented in figure 6 is correctly sorted, it is not suitable for efficient code generation. Fortunately, the action can be simplified by using the sort information just gathered[12]. For instance, the sort variable θ_6 is mapped to the value 6. Any yielder that yields the individual mapped by θ_6 may be replaced by 6. Similarly, any yielder that yields the datum mapped by θ_7 may be replaced by 5.

Once all dependence on θ_6 and θ_7 has been removed, all actions that give θ_6 or θ_7 may be replaced by the action complete. At this point the action has been simplified to the point shown in figure 7. This transformation is analogous to constant propagation found in most conventional compilers.

Since figure 7 includes complex actions involving complete it can be further simplified. The action complete is a unit for the and then combinator. In addition complete then $A = A$ and complete hence $A = A$ are both algebraic identities.

Applying the unit for and then and the algebraic identity involving then the action is further simplified

```
| | allocate [integer]cell
| then
| | bind "M" to the given cell
hence
| | | | store 5 in the cell bound to "M"
| | | then
| | | | give the (sum (6, 5)) [ giving integer]
| | and then
| | | give the integer stored in
| | | the cell bound to "M"
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
```

Figure 8: Simplification of subactions involving complete

```
| bind "M" to [integer]cell$_0$
hence
| | | | store 5 in [integer]cell$_0$
| | | then
| | | | give the (sum (6, 5)) [ giving integer]
| | and then
| | | give the integer stored in [integer]cell$_0$
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
```

Figure 9: Reduced by static allocation of the memory cell

to that shown in figure 8.

The subaction allocate a cell doesn't occur in a dynamic context (i.e. allocation is not within a loop). Therefore, the cell can be allocated statically. By extending Action Semantics to allow cells to be named, we can replace all references to the allocated cell with a named cell. We'll designate the cell's name as [integer]cell$_0$ and replace all yielders that yield the cell with the named cell. Then any actions that give the cell may be replaced by complete. After simplifying subactions involving complete the action is further reduced to the one found in figure 9.

Notice there are no bound to yielders left in the

```
| | | store 5 in [integer]cell$_0$
| | then
| | | give the (sum (6, 5)) [ giving integer]
| and then
| | give the integer stored in [integer]cell$_0$
then
| give the (product (given integer#1,
| given integer#2)) [ giving integer]
```

Figure 10: After binding elimination

action. If there were, the source language and program would not be statically bound! Since there are no bound to yielders left, any occurrences of bind to actions may be replaced by complete and the appropriate identities may be used to reduce the action once again. In this example that results in the action found in figure 10.

The action in figure 10 is suitable for efficient code generation. The subactions resemble assembly language instructions. The transients that are left (θ_{20}, θ_{32}, and θ_{33} in the annotated action) represent the contents of the operand stack in a stack-oriented machine or the contents of symbolic registers in a register-oriented machine. Code generation is a straightforward translation of these *low-level* actions into the target language.

5 Conclusion

This paper has shown that sort inference can be applied to actions to eliminate transients and bindings from statically bound languages. Additionally, the *concat* derived sort inference operation is presented to support record sort inference over tuples. An example of action transformation is presented, demonstrating that transformed actions are, in a sense, *low-level* actions that can be directly translated into efficient code of the target language.

Future work in this area includes studying the mapping of residual transients, those transients left after transient elimination, to symbolic registers in a register-oriented machine. Another topic of research is applying sort inference to Action Semantics descriptions of language for verification and possible simplification of language descriptions.

A Calculator Action Semantics

A.1 Sorts

(1) bindable = [integer]cell

A.2 Semantic Equations

- calc _ :: Program → Act

```
(1)  calc ⟦ E:Expr "=" ⟧ =
         | | allocate an [integer]cell
         | then
         | | bind "M" to the given cell
         hence
         | evaluateExpr E .
```

- evaluateExpr _ :: Expr → Act

```
(2)  evaluateExpr ⟦ E:Expr "+" T:Term ⟧ =
         | | evaluateExpr E
         | and then
         | | evaluateTerm T
         then
         | give the (sum of (the given integer#1,
         | the given integer#2)) [giving an integer]

(3)  evaluateExpr ⟦ E:Expr "-" T:Term ⟧ =
         | | evaluateExpr E
         | and then
         | | evaluateTerm T
         then
         | give the (difference of (the given integer#1,
         | the given integer#2)) [giving an integer]

(4)  evaluateExpr ⟦ T:Term ⟧ =
         evaluateTerm T
```

- evaluateTerm _ :: Term → Act

```
(5)  evaluateTerm ⟦ T:Term "*" F:Factor ⟧ =
         | | evaluateTerm T
         | and then
         | | evaluateFactor F
         then
         | give the (product of (the given integer#1,
         | the given integer#2)) [giving an integer]

(6)  evaluateTerm ⟦ T:Term "/" F:Factor ⟧ =
         | | evaluateTerm T
         | and then
         | | evaluateFactor F
         then
         | give the (quotient of (the given integer#1,
         | the given integer#2)) [giving an integer]

(7)  evaluateTerm ⟦ F:Factor ⟧ =
         evaluateFactor F
```

- evaluateFactor _ :: Factor → Act

```
(8)  evaluateFactor ⟦ G:Storable "S" ⟧ =
         | evaluateStorable G
         then
         | | store the given integer
         | | in the cell bound to "M"
         | and then
         | | give the given integer

(9)  evaluateFactor ⟦ G:Storable ⟧ =
         evaluateStorable G
```

- evaluateStorable _ :: Storable → Act

```
(10) evaluateStorable ⟦ "R" ⟧ =
         give the integer
         stored in the cell bound to "M"

(11) evaluateStorable ⟦ N:Integer ⟧ =
         give N

(12) evaluateStorable ⟦ "(" E:Expr ")" ⟧ =
         evaluateExpr E
```

References

[1] D.F. Brown. *Sort Inference in Action Semantic Specifications*. PhD thesis, Department of Computer Science, University of Glasgow, 1994.

[2] D.F. Brown and D.A. Watt. Sort inference in the actress compiler generator. In *Proceedings of the First International Workshop on Action Semantics*, Edinburgh, Scotland, 1994. BRICS.

[3] O. Danvy. Type-directed partial evaluation. In *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1996.

[4] O. Danvy and R. Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In *Eighth International Symposium on Programming Language Implementation and Logic Programming*, pages 182–497, 1996.

[5] S. Even and D.A. Schmidt. Type inference for action semantics. In *ESOP '90, 3rd European Symposium on Programming, volume 432 of Lecture Notes in Computer Science*, pages 118–133, Berlin, Germany, 1990. Springer-Verlag.

[6] William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE Computer Society, 1998.

[7] S. Liang. A modular semantics for compiler generation. Technical Report TR-1067, Yale University Department of Computer Science, 1995.

[8] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1995.

[9] Peter Mosses. An introduction to action semantics. Technical Report DAIMI PB-370, Aarhus University, Copenhagen, Denmark, 1991.

[10] Peter Mosses. *Action Semantics: Cambridge Tracts in Theoretical Computer Science 26*. Cambridge University Press, 1992.

[11] H. Moura and D.A. Watt. Action transformations in the actress compiler generator. In *Compiler Construction - 5th International Conference CC'94*, 1994.

[12] Hermano Moura. *Action Notation Transformations*. PhD thesis, Department of Computer Science, University of Glasgow, 1993.

[13] Peter Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Computer Science Department, Aarhus University, Denmark, September 1993. URL: `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.

[14] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Adisson Wesley Publishing Company, Inc., New York, NY, 1995.

[15] David Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, Inc., Englewoods Cliffs, New Jersey 07632, 1991.

Representations, Tools, and Techniques for the Complete Integration of Software Development Documents

Ethan V. Munson

July 22, 1998

Note: This document is an abridged version of the original proposal for NSF CAREER Grant CCR-9734102.

Proposal Summary

The proposed research will identify the document representations, software tools, and user interface services needed for the complete integration of all documents produced by the software development process. This work will provide models and tools that make the logical relationships between software documents explicit, thereby allowing them to be queried and analyzed. The proposed research will also bring program source code out of the dated world of eight-bit character streams by allowing multimedia documentation to be embedded within source code documents. This research is significant because it will break down the barriers that currently exist between program source code and the natural language documents that motivate, evaluate, and explain it. These barriers hinder software development by making it difficult to determine whether related documents conform to each other and by restricting in-line documentation of program source code to comments in typewriter-style text.

To meet the aims of this proposal, the investigator will address the following research issues:

- The design of document representations that support the complete interoperability of software documents, regardless of their medium, so that any software document (including source code) can include material in any medium or any relevant fragment of any other software document.
- The design of representations for the logical relationships between software documents that help developers determine whether the contents of related documents are in conformance.
- The design of analytic, visualization and user interface tools that help developers maintain and understand the relationships between their software documents and the extent to which these documents conform to each other.

The success of the research will be evaluated through the implementation of these designs in a development environment for Java programs and all their associated documents. This environment will use World-Wide Web technology to connect the documents but will supplement the standard document and link structure with fine-grained revision control, specialized

link semantics, and tools for analyzing and visualizing the entire document set. The environment will used in its own development and will also be deployed in undergraduate and graduate software engineering courses. Opportunities for deployment in industrial settings will be actively sought.

The proposal's education plan describes a set of activities which will increase enrollment and success of students from traditionally under-represented groups and will improve the university's computer science program at both the undergraduate and graduate levels. The plan includes an innovative curriculum for the GEST Summer Program for middle-school and high-school students based on the World-Wide Web, outreach activities, and new course offerings in software engineering. It integrates recent research results into graduate coursework, as well as using the environment developed by the proposed research at both the undergraduate and graduate levels. In addition, the education plan proposes to enhance the department's graduate program with a complete curriculum on software and systems.

Career Development Plan

I. Research Plan

1 Introduction

The proposed research will identify the document representations, software tools, and user interface services needed for the complete integration of all documents produced by the software development process. This work will provide models and tools that make the logical relationships between software documents explicit, thereby allowing them to be queried and analyzed. The proposed research will also bring program source code out of the dated world of eight-bit character streams by allowing multimedia documentation to be embedded within source code documents. This research is significant because it will break down the barriers that currently exist between program source code and the natural language documents that motivate, evaluate, and explain it. These barriers hinder software development by making it difficult to determine whether related documents conform to each other and by restricting in-line documentation of program source code to comments in typewriter-style text.

The results of this research will be applicable to any software development project that uses a formal process and produces a range of software documents. The growing adoption of software quality standards (such as ISO/FDIS 9000-3) means that the importance of documents in software development will only increase. It is of critical importance that improved tools for managing these documents and the logical relationships between them be developed and adopted. Such tools will also be applicable in any environment where collections of interrelated documents must be maintained and reasoned about, such as large engineering or business projects. Software development is an obvious starting point, but the problem is of general importance.

Programming environments have made great strides in recent years, so that software developers now expect close coordination between tools for editing, compiling, and debugging source code. However, programming environments do not interoperate with the office software suites that are commonly used to produce requirements and design documents, testing and bug reports, and user manuals. Recent research is closing the gap between requirements, design, and source documents through the use of formal specifications, but it remains clear that less-formal documents will be important parts of the software development process for the foreseeable future. Thus, it is important to find ways to better integrate these documents and to help software developers understand and maintain the relationships between them.

To meet the aims of this proposal, the investigator will address the following research issues:

- The design of document representations that support the complete interoperability of software documents, regardless of their medium, so that any software document (including source code) can include material in any medium or any relevant fragment of any other software document.
- The design of representations for the logical relationships between software documents that help developers determine whether the contents of related documents are in conformance.
- The design of analytic, visualization and user interface tools that help developers maintain and understand the relationships between their software documents and the extent to which these documents conform to each other.

This research builds on the principal investigator's prior work on document presentation services [28, 29, 30] and software development environments [10]. The work on document presentation services has produced a portable style sheet system based on a simple and powerful specification language [28] and a model of media that is used to reconfigure this system for applications supporting different media [29]. The principal investigator was one of the architects of the Ensemble integrated software development environment. Ensemble supports both multimedia and program source code documents and has served as a testbed for incremental program analysis techniques [20, 39] and for document presentation services [11, 15, 25, 31]. Ensemble uses a model-view-controller architecture [30, Chapter 3] that allows an arbitrary number of views of the same document and allows non-source-code documents to include other documents, regardless of type. The proposed research will extend this work so that program source code interoperates freely with documents in any medium and the relationships between these documents can be explicitly specified, analyzed, and queried.

Evaluation of the project will be based on testing with end-user populations (primarily students), runtime performance, and technical correctness. The grant project will develop a prototype environment for developing software documents, specifically Java programs and multimedia documents represented using XML [4, 5]. This environment will be tool-based, rather than monolithic, which should ease deployment and testing and will make extensive use of World-Wide Web technology. The primary test population will be students in undergraduate and graduate software engineering courses, who will use the tools in group development projects that require good documentation practices. The tools will also be tested on themselves, as the project will use them in its own development efforts. Finally, the grant project will strive to produce tools that are robust and powerful enough to justify experimental deployment in local industry.

2 Specific Objectives

The proposed research aims to integrate the diverse set of documents produced in developing software. This broad goal requires that the research produce new document representations that improve the interoperability of software documents and make the relationships between the ideas in these documents explicit. Using these new representations, the research must also develop tools and services that make it possible to maintain and understand the relationships between the documents and insure that the contents of the documents are in conformance with each other. These research goals give rise to a number of specific objectives.

The achievement of complete interoperability of software development documents implies that any document can include fragments of any other document, no matter what types of documents are involved. Program source code is the obvious stumbling block, since editors and environments still represent programs as simple text streams. What is needed is a new representation for program source code that allows

- Inclusion of fragments of other documents.
- Active inclusions, which change when the source document changes, and
- The binding of inclusions and multimedia documentation to sections of source code without interfering with the integrated program analysis services (lexing, parsing, static semantic analysis) of the editor or environment.

Given such a representation, new program presentation tools will be required that give programmers flexible control over the way that they view their source code so that the embedded documentation doesn't distract from the development task.

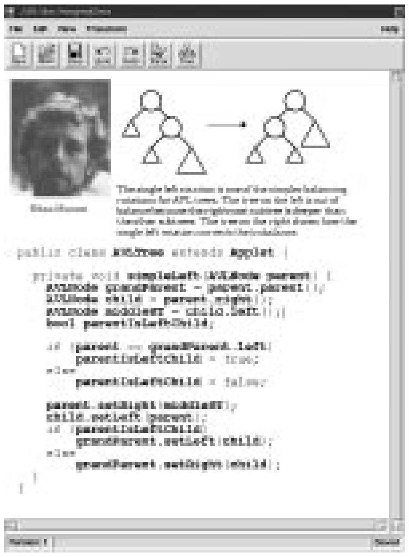
Interoperability between formal and informal documents also requires a uniform model of revision control. While the value of revision control tools for source code is well understood, such tools are not generally available for informal documents, especially non-text documents.

Software development documents have many logical relationships, which may be described by cross-references in the documents' content, but just as often are only implicit. Furthermore, these documents change over time and their logical relationships can also change as a result. The proposed research will develop representations, based on earlier research on hypertext [13, 14, 18], that allow developers to make these relationships (or links) explicit. This goal gives rise to the following objectives:

- A taxonomy of the types of logical relationships between software documents.
- A representation allowing bi-directional, typed links between particular versions of software documents that is compatible with World-Wide Web technology or an enhancement thereof. This representation must be amenable to efficient querying and analysis.

Once it is possible to represent these logical relationships, the proposed research will investigate the tools and services needed to analyze, visualize, and maintain them. Some of the subtasks that this research will accomplish are:

- User interface services that ease the definition of document relationships, so that software developers do not hesitate to specify them.
- Tools which analyze collections of software documents and their relationships for possible conformance problems.
- Visualization tools that help developers understand the level of conformance in a system and locate likely problem areas.



[illegible]

```
public class AVLTree extends Applet {

  private void simpleLeft(AVLNode parent) {
    AVLNode grandParent = parent.parent();
    AVLNode child = parent.right();
    AVLNode middleFT = child.left();
    bool parentIsLeftChild;

    if (parent == grandParent.left)
        parentIsLeftChild = true;
    else
        parentIsLeftChild = false;

    parent.setRight(middleFT);
    child.setLeft(parent);
    if (parentIsLeftChild)
        grandParent.setLeft(child);
    else
        grandParent.setRight(child);
  }
}
```

Figure 1: This screen dump of the Ensemble system illustrates how embedded multimedia documentation could be used to clarify complex algorithms, such as the rotations used to balance AVL trees. A section of Java source code is documented by a 2D graphic showing the rotation along with an explanatory paragraph. While Ensemble has in the past had support for video, this screen dump simulates an explanatory video clip with an included image.

- User interface services that aid in the correction of conflicts between documents.

The research will be evaluated by deploying its tools in undergraduate and graduate software engineering projects and by using the tools in their own development. Other deployment opportunities (in local industry, for example) will be actively sought.

3 Background

3.1 Software documents

The many documents produced by the software development process can be broadly divided into two categories: *formal* and *informal*.

Formal documents include program source code and formal specifications. Their common characteristic is that their syntactic and semantic structure can be determined by analysis of a text stream (with the obvious exception of visual programs). Formal documents are written using ASCII text editors or specialized environments. Even the advanced environments (for example, Ada-Assured [12]) are restricted to text, albeit with font and color variations, and do not support documentation in other media or connections with other software documents. The limitation to textual documentation can prevent programmers from expressing important ideas about their code that are better expressed in other media. Figure 3.1 shows an example where an embedded graphic figure clarifies an algorithm. The fact that programmers cannot link their source code to its supporting documents is just as serious a limitation, since the code is often a direct expression of ideas in those other documents.

All other software documents are informal. Any syntactic or semantic structure they have is either specified directly by the user, obtained from a shared template or form, or is implicit in the natural language content of the document. Examples include requirements documents, design documents, testing and bug reports, and user documentation. Informal documents are commonly produced using commercial office software suites, such as Microsoft Office [23]. MS Office provides extensive interoperability between the different types of documents it supports: any document can include active fragments of other documents. Furthermore, MS Office documents can import a wide variety of multimedia objects. However, these inclusions are not marked with information about their semantics.

3.2 Interoperability

In practice, formal and informal documents do not interoperate. The central problem is that, in formal documents, the text stream is used both for analysis and for presentation [37]. The lexical analysis phase of program analysis requires that the text stream adhere to the language specification, which allows only textual comments. Thus, it is not possible to embed objects composed of arbitrary byte streams (such as compressed images) inside program source code. It is possible to conceive of program editors that search for special comments pointing to pieces of non-text documentation held externally, but there are no examples of such an editor. Such an editor would require a special formatting model to correctly display the non-text documentation.

This is not to say that program presentation has been ignored. In fact, there is a large literature on program pretty-printing. Early work focused on pretty-printing standards for particular languages (see Baecker and Marcus for a summary [1, p. 18]) and on line-breaking algorithms for program statements [17, 24, 32, 33, 40]. More recent work has emphasized specification languages for pretty-printing, such as PPML [16]. All of this work has focused entirely on program source code.

The PI's dissertation research [30] investigated a more general approach to presentation. This research showed there exists a core set of presentation services which can be shared by independent modules for different media within a larger system. This work produced a portable style sheet system, Proteus [11, 28] that can be configured to support different media (text, graphics, video) and is also suitable for program source code. Configuration of Proteus for a new medium (or another application) is performed by writing a specification of the medium's primitive types, dimensions, and presentation attributes [29]. Proteus is also designed to support multiple simultaneous views of the same document in different styles, a mechanism that can be exploited for the production of novel user interfaces without requiring separate formatting services for each view. Proteus is portable and has been used by two systems: the Ensemble software development and multimedia document environment [10] and Multiple Presentation Mosaic [19], a multiple-view browser for the World-Wide Web.[1]

3.3 Document Relationships and Conformance

There are many types of relationships between software development documents. Without claiming to present a complete taxonomy, these are some examples:

[1]Examples of the output of Multiple Presentation Mosaic can be seen on the principle investigator's Web page at `http://www.cs.uwm.edu/faculty/munson`. These demonstrate some of the uses of multiple-view technology.

- The requirements *motivate* the design.
- The design *requires* the implementation.
- A test report *evaluates* the implementation.
- A bug report *complains about* a mismatch between the requirements and the implementation.
- A change to the implementation *responds to* a bug report.
- The user manual *documents* the design and implementation.

In general, these relationships are persistent, lasting days, weeks, or years, but they are not necessarily permanent. Because the documents in a system are dynamic and can be created, altered, and removed, the set of active relationships in a system is also likely to change over time.

Let us consider an imaginary software system whose documents are in perfect harmony with each other. We might say that its documents are *conformant*, because they conform to each other. If we then alter a requirement, such as the number of users to be supported, but make no other change, it becomes possible that the system does not meet its requirements. We might then say that the system's documents are *non-conformant*, because the system's design does not conform to its requirements.

Barring major advances in natural language processing research, completely automatic testing for conformance between software documents will not be possible. However, if the relationships between software documents were explicitly recorded, it might be possible to automate *detection* of *possible* non-conformance. Such automated detection could be used to guide developers to potential problems.

It is important to note that similar relationships exist among source code and specification documents. Programming languages have commands like "include" or "require" that describe a dependency between pairs of files. The difference is that these relationships between formal documents can be found, without any ambiguity, by automated analyses. In fact, relationships like these are an important information source for re-engineering tools [26].

Each of the above document relationships carries with it an implied logical ordering of its documents. For example, testing and bug reports cannot be produced until an implementation is available, and while it is not necessarily the case that requirement documents will be written before designs, there is certainly a logical relationship between them that makes design depend on requirements. Ordered relationships like these have been used for many years to automate efficient compilation [6, 8]. However, these techniques have yet to be applied to informal software documents.

4 Methodology

4.1 Interoperability

The proposed research will develop representations that provide for interoperability among formal and informal documents. The proposed representations will:

- Allow any document to include any fragment of any other document. The included fragment may be from a particular version or may be active, changing as the source document changes.
- Allow the binding of bi-directional, typed relationships to any pair of document fragments.
- Support a uniform model of fine-grained revision control.
- Allow program analysis and compilation services to operate without major modification.

The approach taken by the proposed research will begin with a uniform tree-structured representation for informal documents, such as XML [4, 5], an emerging standard for World-Wide Web documents. Interoperability between tree-structured informal documents is well-understood and can be found in Ensemble [22] and the Grif/Thot system [34].

Then, as proposed in the investigator's earlier work on interoperability [27], a representation that intermixes sections of source code with other sections that contain informal material will be designed. The informal material may either be embedded documentation or may be an inclusion of part of another relevant document. This representation will maintain a clear separation between the source code and informal sections of the document so that program analysis need only traverse the source code sections. The representation will allow the persistent binding of the informal material to the relevant section of the source code, so that the connection between them continues from one editing session to another. Furthermore, it will provide for the definition of persistent selections [14] that will serve as end-points for document relationships.

The proposed representation for source code documents will require the development of a novel formatting model that properly integrates the formal and informal material. The central problem is that automatic pretty-printers operate not from lines of text, but rather from an abstract syntax tree [1, 16]. But in the proposed representation, the leaves of the abstract syntax tree will be intermixed with some other tree-based representation for the informal material. No existing formatter or editor must coordinate between two tree representations, so a new formatting model will be required.

The proposed research will also design a uniform, fine-grained revision control model for software documents, because revision control is a critical element of the software document environments envisioned by the investigator. Figure 2 illustrates how apparent cycles in the relationship graph of software documents are broken when the relationships are defined between versions of the documents.

This work will build on research by Magnusson, Asklund, and Minör [21] on version trees for tree-structured text documents. They showed that version trees are more flexible and semantically correct than the line-oriented revision systems widely used for program source code [35, 36] and are better suited for collaborative software development. Wagner extended this work to integrate version management with program editing and analysis operations in the Ensemble environment [38], but did not provide a persistent representation of the versions. The proposed research will apply version trees to multimedia documents and will provide the persistence missing from Wagner's work.

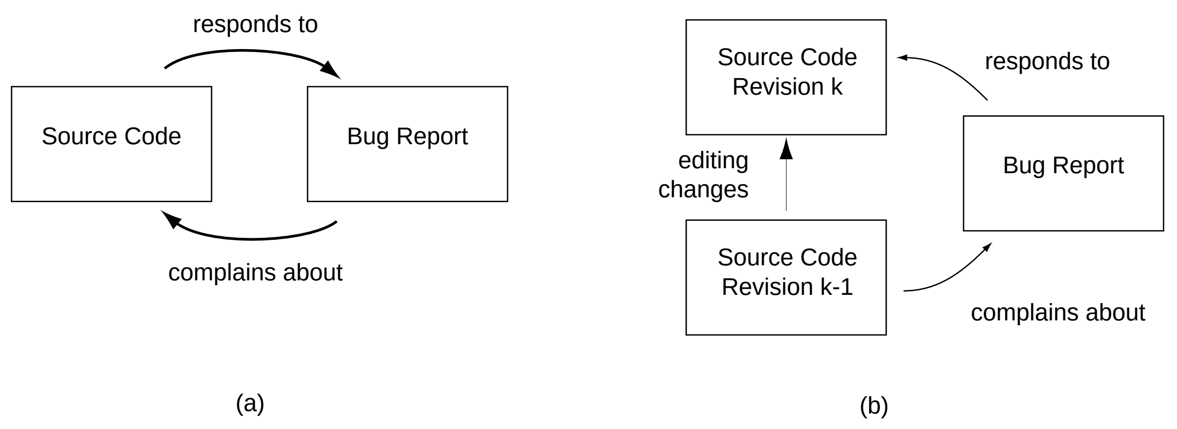


Figure 2: In (a), the relationships between a source code document and a bug report form a cycle, because the code is both cause of the bug report and the response to it. However, the addition of version information in (b) breaks this cycle by making clear that the bug report is complaining about a problem in the version $k-1$ of the source code, while version k of the source code has been edited and responds to the bug report.

4.2 Software Document Relationships

Section 3.3 listed several types of relationships that may exist between software documents. The list is not claimed to be complete, exhaustive, or minimal; it is simply a set of examples.

The proposed research will study relationships between software documents in order to define a taxonomy for them. The types of relationships identified by the taxonomy will then be used to mark links between software documents.

It might be argued that a taxonomy of relationships is unnecessary, that it would be sufficient to simply mark the existence of dependencies between documents without any information about the type of relationship. This argument is misguided because relationship types can convey important information about the semantics of the connection between two documents. Suppose that *complains about* and *comments on* are two relationship types. The fact that *A comments on B* is essentially neutral. In contrast, when *C complains about D*, it is clear that a problem exists. In most systems, complaints require a response, while comments do not.

Using the taxonomy of document relationships, the proposed research will design a representation for links between documents that makes these relationships explicit. This representation will have the following characteristics:

- Links will connect parts of document, rather than entire documents, so that relationships can be defined on a fine-grained basis.
- Since changes to document elements alter their relationships, links will connect specific versions of document elements. This will allow the network of relationships to reflect the dynamic nature of software documents.
- Since a developer may want to follow a link in either direction, the link representation will support traversal in both directions.

- At the same time, the link representation will be compatible with the World-Wide Web convention of embedding links in the source document (which normally makes them uni-directional).
- The representation must allow developers to state whether or not the documents in the relationship are conformant. The simple fact that requirements have changed does not *necessarily* mean that the design is inadequate. It simply means that a developer needs to inspect both documents to determine whether there is a problem.
- The representation must be readily accessible for querying and analysis.

4.3 Analysis, Visualization, and User Interface Services

Once the document and relationship representations have been defined, the proposed research will design and implement services that help developers maintain and understand the relationships between their software development documents. This work will rely on and enhance the experimental testbed described in the next section.

The proposed user interface services will allow the creation, inspection, and removal of links between documents. Developers will also be able to mark links with information about conformance of the documents that each link connects. Other interface services will allow the user to construct queries for types of relationships and levels of conformance. For example, a user may want to find all pairs of non-conforming documents having the *requires* relationship. The responses to these queries can either be reported to the user via a textual interface or by highlighting matching document sections.

In contrast to the user interface services, which will primarily provide information about local relationships, the analysis services will compute properties of the graph of relationships as a whole. Because this information may be complex and hard to understand, visualization tools will be designed and developed, building build on prior research on graphical presentations of large software systems using graphs [26] and compact visual summaries of source code files [2, 7].

The analysis and visualization services will allow developers to answer questions like the following:

- If requirement R is changed, which other documents may require changes?
- When requirement R changed, what changes to other documents had to be made?
- What features of the design have been complained about three or more times?
- How often are design and requirements changes made in response to bug reports?
- How often do bug reports complain about documents that test reports also complained about?

4.4 Evaluation

The concepts explored by the proposed research will be evaluated by implementation in an experimental testbed: an environment for Java programs and their associated informal documents. The environment will use a tool-based design, rather than trying to create a monolithic program, but will include a fairly large editor supporting all types of documents. Furthermore, the environment will be implemented in Java itself, so that over time, the researchers will be able to use and evaluate the tools they are creating.

The Java language will be used because it has a clean syntactic design that eases program analysis and because of its importance to the WWW. Focusing on a single programming language will simplify the system by allowing the use of special-purpose code to handle any tricky problems with incremental program analysis.

Informal documents will be represented using XML [5], an evolving standard for WWW documents. Unlike the HTML standard [3], which defines a single type of document, XML provides a mechanism for defining new types of documents. Furthermore, XML's design is intended to support bi-directional links, but this part of its design is not yet complete [4]. Unlike SGML [9], on which it is based, XML is designed for use in interactive systems.

The core of the environment will be an editor for Java programs and XML documents. The editor's features will include:

- Full interoperability between all document types,
- Integrated parsing and type-checking of Java programs,
- User interface features for managing document relationships,
- World-Wide Web compatibility, and
- Integration with the environment's revision control system.

All other tools and services, including revision control, relationship analysis, and relationship visualization, will be implemented as independent programs.

The experimental testbed will be used for group projects in undergraduate and graduate software engineering classes. Students using the tools will be surveyed about the tools usefulness and usability. More detailed evaluation information will be obtained through debriefings of selected students in these courses and through usability studies with small groups of users. Deployment of the experimental testbed in industrial settings is also planned and will use similar evaluation methods.

Other sources of evaluative information will be:

- Systematic use of mock-up situations to evaluate the design of document and link representations and the taxonomy of document relationships.
- The use of test suites to evaluate the correctness of the environment's tools.
- Performance testing of tools.

References

[1] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs.* Addison-Wesley, Reading, Massachusetts, 1990.

[2] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.

[3] T. Berners-Lee and D. Connolly. *Hypertext Markup Language — 2.0.* World Wide Web Consortium and MIT, June 1995. Internet Draft available from www.w3c.org.

[4] Tim Bray and Steve DeRose. Extensible markup language (xml): Part 1. linking. Available on the World Wide Web at http://www.w3.org/TR/WD-xml-link, April 1997.

[5] Tim Bray and C. M. Sperberg-McQueen. Extensible markup language (xml): Part 1. syntax. Available on the World Wide Web at http://www.w3.org/TR/WD-xml-lang, June 1997.

[6] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions of Programming Languages and Systems*, 12(1):1–25, January 1990.

[7] Stephen G. Eick, Michael C. Nelson, and Jeffery D. Schmidt. Graphical analysis of computer log files. *CACM*, 37(12):50–56, December 1994.

[8] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9:255–265, 1979.

[9] Charles F. Goldfarb, editor. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML).* International Organization for Standardization, Geneva, Switzerland, 1986. International Standard ISO 8879.

[10] Susan L. Graham. Language and document support in software development environments. In *Proceedings of the Darpa '92 Software Technology Conference*, Los Angeles, April 1992.

[11] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 130–138, Tyson's Corner, VA, December 1992. ACM Press.

[12] Inc. Grammatech. Ada-assured home page. Accessible at http://www.grammatech.com, 1997.

[13] Bernard J. Haan, Paul Kahn, Victor A. Riley, James H. Combs, and Norman K. Meyrowitz. IRIS hypermedia services. *CACM*, 35(1):36–51, January 1992.

[14] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. *CACM*, 37(2):30–39, February 1994.

[15] Michael A. Harrison and Vance Maverick. Presentation by tree transformation. In *IEEE COMPCON '97*, February 1997.

[16] INRIA: Centaur Project, Sophia-Antipolis, France. *The PPML Manual*, February 1994. For Version 1.3 of Centaur. Available by ftp from babar.inria.fr in directory `pub/croap/bertot`.

[17] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software—Practice & Experience*, 11(11):1119–1184, November 1982.

[18] John J. Leggett and John L Schnase. Viewing dexter with open eyes. *CACM*, 37(2):76–86, February 1994.

[19] Hong Liu. Multiple Presentation Mosaic. Master's thesis, University of Wisconsin-Milwaukee, May 1996.

[20] William Harry Maddox, III. *Incremental Static Semantic Analysis*. PhD thesis, University of California, Berkeley, January 1997.

[21] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 33–41. ACM Press, December 1993.

[22] Vance Maverick. *Presentation by Tree Transformation*. PhD thesis, University of California, Berkeley, 1997.

[23] Microsoft, Inc., Redmond, Washington, USA. *Microsoft Office 4.2*, 1995.

[24] Martin Mikelsons. Prettyprinting in an interactive programming environment. Technical Report RC 8756, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, March 1981.

[25] Alok Mittal. SHILPĒ: A presentation system for ensemble. Master's thesis, University of California, Berkeley, California, December 1995.

[26] H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 88–98, Tyson's Corner, VA, December 1992. ACM Press.

[27] Ethan V. Munson. Interoperability of software documents. In *Workshop on Software Engineering and Human-Computer Interaction: Joint Research Issues*, pages 153–162, May 1994. Workshop pre-prints.

[28] Ethan V. Munson. A new presentation language for structured documents. *Electronic Publishing: Origination, Dissemination, and Design*, 8:125–138, September 1995. Originally presented at EP96, the Sixth International Conference on Electronic Publishing, Document Manipulation, and Typography, Palo Alto, CA, September 1996.

[29] Ethan V. Munson. Toward an operational theory of media. In *Proceedings of the Third International Workshop on Principles of Document Processing*. Springer-Verlag, Palo Alto, CA, September 1996. To be published as part of the Lecture Notes in Computer Science series.

[30] Ethan Vincent Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment.* PhD dissertation, University of California, Berkeley, December 1994. Also available as UC Berkeley Computer Science Technical Report UCB/CSD-94-833.

[31] Kannan Muthukkaruppan. SPINE, a synthesizer for practical incremental evaluators. Master's thesis, University of California, Berkeley, CA, May 1994.

[32] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[33] William W. Pugh and Steven J. Sinofsky. A new language-independent prettyprinting algorithm. Technical Report TR 87-808, Dept. of Computer Science, Cornell University, Ithaca, NY, January 1987.

[34] Vincent Quint and Irène Vatton. Combining hypertext and structure documents in grif. In D. Lucarella, editor, *Proceedings of ECHT '92*, pages 23–32, Milan, December 1992. ACM Press.

[35] M. J. Roekind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[36] W. F. Tichy. RCS — a system for revision control. *Software: Practice and Experience*, 15(7):637, July 1985.

[37] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37:431–466, 1992.

[38] Tim A. Wagner and Susan L. Graham. Integrating incremental analysis with version management. In *Proceedings of the Fifth European Software Engineering Conference*, 1995.

[39] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 31–43, June 1997.

[40] Richard C. Waters. XP: A Common Lisp pretty printing system. A.I. Memo 1102a, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, August 1989. Also appears in edited form as Chapter 27 of *Common Lisp: The Language, second ed.*