Program Specialization via Program Slicing

Thomas Reps and Todd Turnidge
University of Wisconsin

Abstract

This paper concerns the use of program slicing to perform a certain kind of program-specialization operation. The specialization operation that slicing performs is different from the specialization operations performed by algorithms for partial evaluation, supercompilation, bifurcation, and deforestation. In particular, we present an example in which the specialized program that we create via slicing could not be created as the result of applying partial evaluation, supercompilation, bifurcation, or deforestation to the original unspecialized program.

Specialization via slicing also possesses an interesting property that partial evaluation, supercompilation, and bifurcation do not possess: The latter operations are somewhat limited in the sense that they support tailoring of existing software only according to the ways in which parameters of functions and procedures are used in a program. Because parameters to functions and procedures represent the range of usage patterns that the designer of a piece of software has *anticipated*, partial evaluation, supercompilation, and bifurcation support specialization only in ways that have already been "foreseen" by the software's author. In contrast, the specialization operation that slicing supports permits programs to be specialized in ways that do not have to be anticipated by the author of the original program.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques – *computer-aided software engineering (CASE), programmer workbench*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance – *restructuring*; D.2.m [**Software Engineering**]: Miscellaneous – *reusable software*; D.3.2 [**Programming Languages**]: Language Classifications – *applicative languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *procedures, functions, and subroutines*, *recursion*; D.3.4 [**Programming Languages**]: Processors; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs – *functional constructs*, *program and recursion schemes*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: program projection, program slicing, program specialization, projection function, partially needed structures

1. Introduction

Program slicing is an operation that identifies semantically meaningful decompositions of programs, where the decompositions consist of elements that are not textually contiguous [37, 25, 12]. Program slicing has been studied primarily in the context of imperative programming languages [32]. In such languages, slicing is typically carried out using *program dependence graphs* [18, 25, 6, 12]. There are two kinds of slices of imperative programs: (i) a *backward slice* of a program with respect to a set of program elements S consists of all program elements that might affect (either directly or transitively) the values of the variables used at members of S; (ii) a *forward slice* with respect to S consists of all program elements that might be affected by the computations performed at members of S [12]. For example, a C program and one of its backward slices is shown in Figure 1. Slicing—and subsequent manipulation of slices—shows great promise for helping with many software-engineering problems: It has applications in program understanding,

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

Authors' address: Computer Sciences Department, University of Wisconsin–Madison, 1210 W. Dayton St., Madison, WI 53706.
E-mail address: {reps,turnidge}@cs.wisc.edu.

```
static int add(a, b)
int a, b;
{
  return(a + b);
}

void main()
{
  int sum, i;

  sum = 0;
  i = 0;
  while (i < 10) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  printf("sum = %d\n", sum);
  printf("i = %d\n", i);
}
```

```
static int add(a, b)
int a, b;
{
  return(a + b);
}

void main()
{
  int [ ] i;

  [ ]
  i = 0;
  while (i < 10) {
    [ ]
    i = add(i, 1);
  }
  [ ]
  printf("i = %d\n", i);
}
```

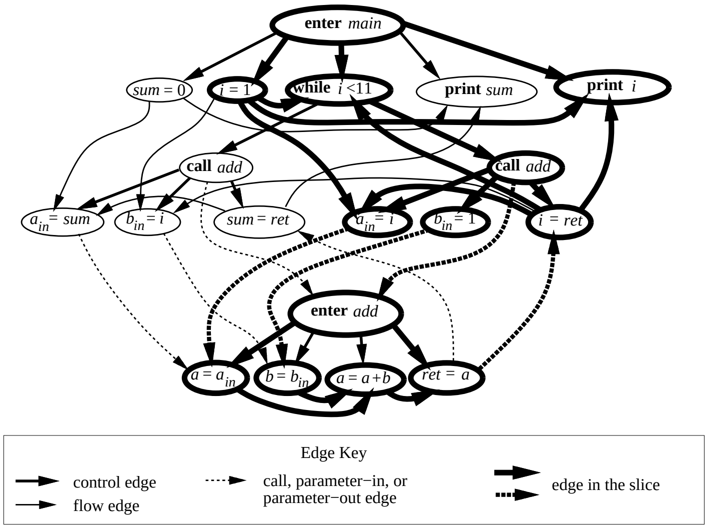

Figure 1. A C program, the slice of the program with respect to the statement *printf("i = %d\n", i)*, and the program's system dependence graph. In the slice, the starting point for the slice is shown in italics, and the empty boxes indicate where program elements have been removed from the original program. In the dependence graph, the edges shown in boldface are the edges in the slice.

maintenance [8, 9], debugging [21], testing [3, 2], differencing [11,13], reuse [24], and merging [11].

This paper concerns the use of slicing to perform program specialization, and how slicing-based specialization relates to partial evaluation and related operations. The contributions of the paper can be summarized as follows:

- We show that the specialization operation that slicing performs is ***different from the specialization operations performed by partial evaluation, supercompilation, bifurcation, and deforestation***. In particular, there are situations in which the specialized programs that we create via slicing could not be created as the result of applying partial evaluation, supercompilation, bifurcation, or deforestation to the original unspecialized program.
- In order to study the relationship between slicing and partial evaluation in a simplified setting, we consider the problem of ***slicing functional programs***. In particular, we identify two different goals for what we mean by "slicing a functional program" and give slicing algorithms that correspond to each of them.
- We adapt techniques from shape analysis [15], strictness analysis [34], and program bifurcation [23] so that ***our slicing algorithms can handle certain kinds of heap-allocated data structures*** (*e.g.*, lists, trees, and dags). This represents a contribution to the slicing literature: By allowing programs to be sliced with respect to "partially needed structures", our techniques can carry out non-trivial slices of programs that make use of heap-allocated data structures.
- We present ***a re-examination of certain aspects of program bifurcation*** in terms of the machinery developed for slicing functional programs.

The remainder of the paper is organized as follows: Section 2 demonstrates specialization via slicing, and shows that slicing performs a different kind of specialization operation from those performed by partial evaluation, supercompilation, bifurcation, and deforestation. Section 3 presents our methods for slicing functional programs. Section 4 compares the semantic issues that arise in specialization via partial evaluation versus specialization via slicing. Section 5 describes how our techniques relate to work on program bifurcation. Section 6 discusses related work. Section 7 presents some concluding remarks.

2. Program Specialization: Program Slicing Versus Partial Evaluation

In some circles, the terms “program specialization” and “partial evaluation” are treated almost as synonyms, although sometimes “program specialization” carries the nuance of expressing a broader perspective that encompasses a number of kindred techniques, such as “generalized partial evaluation” [7], “supercompilation” [33], “bifurcation” [23], and “deforestation” [35]. However, this overlooks an often unappreciated fact, namely that program slicing can also be used to perform a kind of program specialization—and one that is different from the kinds of specializations that partial evaluation and its close relatives are capable of performing.

Example. In the context of imperative programs, this phenomenon is illustrated by the C program shown in the first column of Figure 2 and the two slices shown in the second and third columns. The program in the first column is a scaled-down version of the UNIX word-count utility. It scans a file, counts the number of lines and characters in the file, and prints the results. Thus, this program implements the same action that would be obtained by invoking the UNIX word-count utility with the *–lc* flag (*i.e.*, *wc –lc*). However, unlike the actual UNIX word-count utility, procedure line_char_count is *not* parameterized by a second argument to allow the caller to choose which of the output quantities are to be printed.

Equivalent to *wc –lc*	Equivalent to *wc –c*	Equivalent to *wc –l*
void line_char_count(FILE *f) { int lines = 0; int chars; BOOLEAN eof_flag = FALSE; int n; extern void scan_line(FILE *f, BOOLEAN *bptr, int *iptr); scan_line(f, &eof_flag, &n); chars = n; while (eof_flag == FALSE) { lines = lines + 1; scan_line(f, &eof_flag, &n); chars = chars + n; } printf("lines = %d\n", lines); printf("chars = %d\n", chars); }	void char_count(FILE *f) { int chars; BOOLEAN eof_flag = FALSE; int n; extern void scan_line(FILE *f, BOOLEAN *bptr, int *iptr); scan_line(f, &eof_flag, &n); chars = n; while (eof_flag == FALSE) { scan_line(f, &eof_flag, &n); chars = chars + n; } *printf("chars = %d\n", chars);* }	void line_count(FILE *f) { int lines = 0; BOOLEAN eof_flag = FALSE; extern void scan_line2(FILE *f, BOOLEAN *bptr); scan_line2(f, &eof_flag); while (eof_flag == FALSE) { lines = lines + 1; scan_line2(f, &eof_flag); } *printf("lines = %d\n", lines);* }

Figure 2. A scaled-down version of the UNIX word-count utility and two of its slices. The program in column one counts the number of lines and characters in a file and prints the results. Unlike the actual UNIX word-count utility, procedure line_char_count is *not* parameterized by a second argument to allow the caller to vary the program’s behavior (*i.e.*, to choose which of the two possible output quantities are to be printed).

The procedure char_count shown in column two of Figure 2 is the (backward) slice of line_char_count with respect to the statement *printf("chars = %d\n", chars)*. This slice implements the same action that would be obtained by invoking *wc –c*: it scans a file and counts only characters. The procedure line_count shown in column three is the slice of line_char_count with respect to the statement *printf("lines = %d\n", lines)*. Line_count implements the same action that would be obtained by invoking *wc –l*: it scans a file and counts only lines.

Had the implementor of line_char_count foreseen the need to parameterize the procedure with a second parameter to allow the caller to vary the procedure's output behavior (call this hypothetical procedure "line_char_count+"), then char_count and line_count could have been obtained by partially evaluating line_char_count+ with respect to *–c* and *–l*, respectively. However, ***given the unparameterized line_char_count of Figure 2, partial evaluation does not provide a way to obtain procedures char_count and line_count***. Procedure line_char_count has only the *single* parameter f, and thus there is no opportunity for partial evaluation with respect to a full parameter value. We can perform *full evaluation* (if f's value is provided) or *no evaluation* (if f's value is withheld). Furthermore, none of the information in parameter f controls whether the output consists of just the character count, just the line count, or both the character count and line count together. Thus, even if a partially static value were supplied for f, it would not provide the right kind of information that a partial evaluator would need to create char_count and line_count. □

What this example shows is that slicing performs a different program-specialization operation than that obtained via partial evaluation (or the other forwards-oriented specialization operations). The two approaches are actually *complementary*: Partial evaluation and its forwards-oriented relatives take information known at the *beginning* of a program and push it *forward*; slicing takes a demand for information at the *end* of a program and pushes it *backward*.

In addition, slicing-based specialization has another characteristic that sets it apart from the forwards-oriented specialization operations. The parameters to functions and procedures define the range of usage patterns that the designer of a piece of software has anticipated. This imposes some limitations on partial evaluation, supercompilation, and bifurcation in the sense that they support tailoring of existing software only in ways that have already been "foreseen" by the software's author. In contrast, slicing-based specialization permits programs to be specialized in ways that do not have to be anticipated—via parameterization—by the writer of the original program.

3. Slicing Functional Programs

To elucidate further the relationship between partial evaluation and program slicing, in this section of the paper we study the problem of how to slice functional programs. By considering slicing in a simplified context—and in particular one in which the majority of work on partial evaluation has been carried out—we can better understand the relationship between partial evaluation and slicing, including both common and complementary aspects.

There are other benefits as well: Past work on shape analysis [15], strictness analysis [34], and program bifurcation [23] for functional programs has developed techniques to handle certain kinds of heap-allocated data structures (*e.g.*, lists, trees, and dags). We will use similar techniques to formulate a slicing algorithm that can handle programs that use (heap-allocated) lists, trees, and dags. The slicing algorithm can perform non-trivial slices, where the goal is to satisfy a demand for a "partially needed structure". While a few previous papers on slicing have studied the impact of dependences carried through heap-allocated data structures [10,28], this is an issue that (to date) has been treated peripherally in the slicing and dependence-graph literature.

The slicing algorithm will be formulated for a first-order LISP-like functional language that has the constructor and selector operations NIL, CONS, CAR, and CDR for manipulating heap-allocated data (*i.e.*, lists and dotted pairs), together with appropriate predicates (EQUAL, ATOM, and NULL), but no operations for destructive updating (*e.g.*, RPLACA and RPLACD). The constructs of the language are

x_i	(CONS e_1 e_2)
(QUOTE c)	(IF e_1 e_2 e_3)
(CAR e_1)	(CALL f e_1 $\cdots$ e_k)
(CDR e_1)	(OP op e_1 e_2)
(ATOM e_1)	(DEFINE ($main$ x_1 $\cdots$ x_k) e_{main})
(NULL e_1)	(DEFINE (f x_1 $\cdots$ x_k) e_f)
(EQUAL e_1 e_2)	

A program is a list of function definitions, with a distinguished top-level goal function, named *main*, that cannot be called by any of the other functions. We assume that the distinguished atom “NIL” is used for terminating lists, and that there is also a special empty-tree value (different from NIL), denoted by “?”.

3.1. Projection Functions and Regular Tree Grammars

Our approach to slicing functional programs involves formulating the problem as one of symbolically composing the program to be sliced with an appropriate *projection function* π_{main}. A projection function π is an idempotent function (*i.e.*, $\pi \circ \pi = \pi$) that approximates the identity function (*i.e.*, $\pi \sqsubseteq id$). A projection function can be used to characterize what information should be “discarded” and what information should be “retained” from the value that a function computes. Thus, projection function π_{main} represents a demand for a “partially needed structure”. In the nomenclature used in the slicing literature, π_{main} is called the *slicing criterion*.

Example. Let ID be the identity function $\lambda x.x$ and let Ω be $\lambda x.?$, the function that always returns the empty-tree value. If f and g are two projection functions, let $\langle f.g \rangle$ denote the projection function on pairs such that

$$\langle f.g \rangle (x, y) = (f(x).g(y)).$$

Suppose we want to slice a functional version of the line_char_count program from Figure 2, say LineCharCount, where LineCharCount takes a string and returns a pair consisting of the line count and the character count. Then a program LineCount that only counts lines can be defined by

$$\text{LineCount} =_{df} \langle \text{ID}.\Omega \rangle \circ \text{LineCharCount}.$$

That is, program LineCount can be created by slicing LineCharCount with respect to the slicing criterion $\langle \text{ID}.\Omega \rangle$. □

The challenge is to devise a slicing algorithm that, given a program p and a projection function π, creates a program that behaves like $\pi \circ p$. The slicing algorithm will create the composed program by symbolically pushing π backwards through the body of p and simplifying the function body in appropriate ways.

There are a number of techniques developed for partial evaluation that are related to this goal. For example, projection functions have been used in binding-time analysis for partial evaluation in the presence of partially static structures [19,22]. However, for slicing, we need to propagate projection functions backwards—from function outputs to function arguments. Thus, the slicing problem has similarities with the algorithms that propagate projection functions backwards to perform strictness analysis of lazy functional languages [14,34].

Instead of the fixed, finite domain of projection functions used in [14] and [34], we will use regular tree grammars (see below), which can be viewed as (representations of) projection functions. Specifically, we will use the variant of regular tree grammars that Mogensen used in his work on program bifurcation [23].[1]

A finite tree (or dag) T can be treated formally as a finite prefix-closed set of strings $L(T)$, where $L(T)$ consists of the set of access paths in T. $L(T)$ consists of strings that are either of the form $s_1.s_2. \cdots .s_k$, where the s_i are selectors, or of the form $s_1.s_2. \cdots .s_k.a$, where a is an atom. There is a further "tree constraint" on $L(T)$, which is that if $s_1.s_2. \cdots .s_k.a \in L(T)$, then $L(T)$ cannot contain any string of the form $s_1.s_2. \cdots .s_k.x$, where x is either a selector or an atom. The special empty-tree value "?" corresponds to the empty set of access paths (*i.e.*, $L(?) = \emptyset$).

A projection function π on tree- (or dag-)structured data can also be treated formally as a prefix-closed set of strings $L(\pi)$. However, we do not insist that $L(\pi)$ have the "tree constraint", nor must $L(\pi)$ necessarily be finite. Given a tree T and projection function π, the application of π to T satisfies

$$\pi(T) = T' \text{ such that } L(T') = L(\pi) \cap L(T).$$

For our purposes, we need projection functions that correspond to *infinite* sets of paths. To represent each projection function in a *finite* way, we use a certain kind of regular tree grammar.

A regular tree grammar permits defining a (possibly infinite) collection of trees that share certain structural properties in common. Each production in a regular tree grammar has one the following forms, where A and B stand for either $\top$, $\perp$, or a set of nonterminal names:

$$\begin{array}{l} N \rightarrow \top \\ N \rightarrow \perp \\ N \rightarrow \circ \\ N \rightarrow \langle A.B \rangle \\ N \rightarrow \circ \mid \langle A.B \rangle \end{array}$$

$\circ$ is a special symbol that stands for any atom; $\top$ denotes the set of all trees; $\perp$ denotes the set consisting of the empty tree; the symbols "$\langle$" and "$\rangle$" denote the pairing of trees.

Example. The following regular tree grammar specifies the collection of all odd-length finite lists in which all elements in odd positions along the list are atoms:

$$\begin{array}{l} \mathit{OddList} \rightarrow \langle \{ \mathit{Atom} \}.\{ \mathit{EvenList} \} \rangle \\ \mathit{Atom} \rightarrow \circ \\ \mathit{EvenList} \rightarrow \circ \mid \langle \top.\{ \mathit{OddList} \} \rangle \end{array}$$

(Since we do not distinguish NIL from the other atoms, this actually specifies lists terminated by any atom. This is only a matter of convenience; it would be possible to extend the class of grammars introduced above to treat NIL separately from the other atoms.) □

Given a regular tree grammar G, each nonterminal of G can be viewed as a generator of a set of trees. However, we have no direct use for this view, and instead view each nonterminal as a generator of a (possibly infinite) prefix-closed set of access paths. (Actually, this set of paths is the union of the sets of access paths in the aforementioned set of trees.)

To define the access paths V_N associated with a nonterminal N, we treat the grammar as a collection of equations on prefix-closed sets of strings. For example, if the right-hand side for nonterminal N is $\top$, then we have the equation $V_N = \mathit{Paths}$, where *Paths* is the universe of access paths. If the right-hand side for nonterminal N is $\perp$, then we have the equation $V_N = \emptyset$. If the right-hand side for nonterminal N is

---

[1] In Mogensen's work, regular tree grammars are used as "shape descriptors", to summarize the possible "shapes" that heap-allocated structures in a program can take on, as well as (representations of) projection functions. We will also use regular tree grammar in both of these ways; regular tree grammars are used as shape descriptors in Section 3.4.

$$\circ \mid \langle \{ A_1, \ldots, A_a \}.\{ B_1, \ldots, B_b \} \rangle,$$

the equation for N is

$$V_N = \text{ATOM} \cup \mathbf{cons}(V_{A_1} \cup \cdots \cup V_{A_a}, V_{B_1} \cup \cdots \cup B_{A_b}).$$

In this equation, ATOM is the set of atoms, and **cons** is a set-valued function defined as follows:

$$\mathbf{cons} =_{df} \lambda S_1.\lambda S_2.\{ \text{hd} \} \cup \{ \text{tl} \} \cup \{ \text{hd}.p_1 \mid p_1 \in S_1 \} \cup \{ \text{tl}.p_2 \mid p_2 \in S_2 \}.$$

The language of access paths associated with a nonterminal N is the value of V_N in the least solution to the grammar's equations.

Because a regular tree grammar has a finite number of productions, it provides a way to give a finite presentation of a collection of projection functions: Every nonterminal N corresponds to an associated projection function π_N, where $L(\pi_N) = V_N$.

During context analysis (see Section 3.2), we create an appropriate projection function for each point in the program. This requires us to be able to perform certain operations on projection functions. However, during context analysis all "manipulations of projection functions" are done indirectly, by performing (syntactic) manipulations on right-hand sides of regular-tree-grammar productions. In particular, Figure 3 defines the operator "join", denoted by $\sqcup$, which combines two regular-tree-grammar right-hand sides.

For any given regular tree grammar, we will occasionally use a nonterminal as a synonym for its right-hand side. In addition, we allow $\top$ to be replaced with $\circ \mid \langle \top.\top \rangle$ whenever convenient or necessary.

Remark. The regular tree grammars defined above are not the only form of regular tree grammars that have been defined. For example, one alternative definition has only singleton nonterminals in each branch of each pair that occurs on the right-hand side of a grammar rule, but allows there to be more than one such pair in each right-hand side [15]. This yields a more powerful tree-definition formalism. A feeling for the kind of information that is lost by using sets of nonterminals can be obtained by considering how the join of two right-hand sides is handled under the two approaches:

$$\langle N_1.N_2 \rangle \sqcup \langle N_3.N_4 \rangle = \langle N_1.N_2 \rangle \mid \langle N_3.N_4 \rangle$$

(a) Jones and Muchnick [15]

$$\langle \{ N_1 \}.\{ N_2 \} \rangle \sqcup \langle \{ N_3 \}.\{ N_4 \} \rangle = \langle \{ N_1, N_3 \}.\{ N_2, N_4 \} \rangle$$

(b) Mogensen [23]

Approach (a) forms a right-hand side with multiple alternatives; this preserves the links between N_1 and N_2 and between N_3 and N_4. In approach (b), a single right-hand-side pair is formed that has a set of non-

---

$$X \otimes Y = \begin{cases} \top & \text{if } X = \top \text{ or } Y = \top \\ X & \text{if } Y = \bot \\ Y & \text{if } X = \bot \\ X \cup Y & \text{otherwise} \end{cases}$$

$$\begin{gathered}
\bot \sqcup x = x \\
\top \sqcup x = \top \\
\circ \sqcup \circ \mid \langle A.B \rangle = \circ \mid \langle A.B \rangle \\
\circ \sqcup \langle A.B \rangle = \circ \mid \langle A.B \rangle \\
\langle A.B \rangle \sqcup \langle C.D \rangle = \langle A \otimes C.B \otimes D \rangle \\
\circ \mid \langle A.B \rangle \sqcup \langle C.D \rangle = \circ \mid \langle A \otimes C.B \otimes D \rangle \\
\circ \mid \langle A.B \rangle \sqcup \circ \mid \langle C.D \rangle = \circ \mid \langle A \otimes C.B \otimes D \rangle
\end{gathered}$$

Figure 3. Definition of the join operator $\sqcup$ for combining two regular-tree-grammar right-hand sides. (Join is also commutative; *i.e.*, $x \sqcup y = y \sqcup x$.)

terminals in each arm; this breaks the links between N_1 and N_2 and between N_3 and N_4. The tree descriptions are sharper with regular tree grammars of type (a): With type-(a) grammars, nonterminals N_1 and N_4 can never occur simultaneously, whereas type-(b) grammars permit N_1-trees to be paired with N_4-trees.

However, the way we have defined the correspondence between a nonterminal and its projection function is based on the set of *access paths* of a set of trees (and not on the set of trees *per se*). That is, our intention is to use regular tree grammars as a way to define sets of access paths, one set per nonterminal. For this purpose, type-(a) grammars are no sharper than type-(b) grammars. In addition, it is computationally more expensive to use and manipulate type-(a) grammars [23]. (To take advantage of the sharper information of type-(a) grammars, we would have to switch to a scheme in which each projection function corresponds to a *set of sets of access paths*.) □

3.2. Context Analysis via Regular Tree Grammars

For slicing, we are concerned with information that *might be needed* to compute some portion of the *desired part* of function *main*'s return value, where the "desired part" of the return value is characterized by the "slicing criterion", namely projection function π_{main}. Projection function π_{main} represents a "contract" to limit attention to the portions—if any—of *main*'s return value that lie on the access paths in $L(\pi_{main})$. Thus, in the slice we need only retain the parts of the original program that could contribute to a portion of *main*'s return value that lies on an access path in $L(\pi_{main})$. To identify these parts of the program, the slicing algorithm will propagate π_{main} backwards through the body of the program and simplify the program's subexpressions in appropriate ways.

The goal of slicing is to create a program q such that, on all inputs, q returns the same value as $\pi_{main} \circ p$ applied to the same input. That is,

$$[\![q]\!] = \pi_{main} \circ [\![p]\!] \qquad (\dagger)$$

where $[\![\cdot]\!]$ represents the meaning function of the language. Because projection function π_{main} is idempotent, we have

$$\begin{aligned} \pi_{main} \circ [\![q]\!] &= \pi_{main} \circ (\pi_{main} \circ [\![p]\!]) \\ &= (\pi_{main} \circ \pi_{main}) \circ [\![p]\!] \\ &= \pi_{main} \circ [\![p]\!] \\ &= [\![q]\!] \end{aligned}$$

Thus, strictly speaking, the return value of q's *main* function should contain *no* portions that lie outside of the access paths in $L(\pi_{main})$. In certain situations, we will relax condition (†) to $[\![q]\!] \sqsupseteq \pi_{main} \circ [\![p]\!]$ and (safely) let q's *main* function return a value that *does* have portions that lie outside of the access paths in $L(\pi_{main})$.

Slicing criterion π_{main} is specified by giving a regular tree grammar. For example, the *OddList/EvenList/Atom* grammar given earlier is an example of the kind of description that could be furnished as input to the slicing procedure.

The process of propagating π_{main} backwards through the program is carried out by a *context-analysis* phase. Context analysis is concerned with describing, for each subexpression n of the program, what parts of the values computed by n are possibly needed in *main*'s return value [14,34]. In our case, context analysis creates a regular tree grammar whose nonterminals correspond to the interior points in the program's expression tree, thereby associating each subexpression of the program with (a representation of) a projection function.

The context analysis is specified in Figure 4, which gives schemas for generating one or more equations at each node of the program's expression tree. In general, these schemas generate a collection of mutually recursive equations over two sets of variables: variables of the form Context(n), where n is an interior point

Form of expression	Equations associated with expression
$n : x_i$	$\mathrm{Context}(x_i) = \mathrm{Context}(n)$
$n : (\mathrm{QUOTE}\ c)$	---
$n : (\mathrm{CAR}\ n_1 : e_1)$	$\mathrm{Context}(n_1) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \langle \{\, n \,\}.\perp \rangle$
$n : (\mathrm{CDR}\ n_1 : e_1)$	$\mathrm{Context}(n_1) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \langle \perp.\{\, n \,\} \rangle$
$n : (\mathrm{ATOM}\ n_1 : e_1)$	$\mathrm{Context}(n_1) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \circ \mid \langle \perp.\perp \rangle$
$n : (\mathrm{NULL}\ n_1 : e_1)$	$\mathrm{Context}(n_1) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \circ \mid \langle \perp.\perp \rangle$
$n : (\mathrm{EQUAL}\ n_1 : e_1\ n_2 : e_2)$	$\mathrm{Context}(n_1) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \top$ $\mathrm{Context}(n_2) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \top$
$n : (\mathrm{CONS}\ n_1 : e_1\ n_2 : e_2)$	$\mathrm{Context}(n_1) = \begin{cases} \top & \text{if } \mathrm{Context}(n) = \top \text{ or } \circ \mid \langle \top.B \rangle \text{ or } \langle \top.B \rangle \\ \perp & \text{if } \mathrm{Context}(n) = \perp \text{ or } \circ \text{ or } \circ \mid \langle \perp.B \rangle \text{ or } \langle \perp.B \rangle \\ \bigsqcup_{A_i \in A} A_i & \text{if } \mathrm{Context}(n) = \circ \mid \langle A.B \rangle \text{ or } \langle A.B \rangle \end{cases}$ $\mathrm{Context}(n_2) = \begin{cases} \top & \text{if } \mathrm{Context}(n) = \top \text{ or } \circ \mid \langle B.\top \rangle \text{ or } \langle B.\top \rangle \\ \perp & \text{if } \mathrm{Context}(n) = \perp \text{ or } \circ \text{ or } \circ \mid \langle B.\perp \rangle \text{ or } \langle B.\perp \rangle \\ \bigsqcup_{A_i \in A} A_i & \text{if } \mathrm{Context}(n) = \circ \mid \langle B.A \rangle \text{ or } \langle B.A \rangle \end{cases}$
$n : (\mathrm{IF}\ n_1 : e_1\ n_2 : e_2\ n_3 : e_3)$	$\mathrm{Context}(n_1) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \circ \mid \langle \perp.\perp \rangle$ $\mathrm{Context}(n_2) = \mathrm{Context}(n)$ $\mathrm{Context}(n_3) = \mathrm{Context}(n)$
$n : (\mathrm{CALL}\ f\ n_1 : e_1\ \cdots\ n_k : e_k)$	$\mathrm{Context}(n_i) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \mathrm{ContextEnv}(x_i)$, where f is defined by $(\mathrm{DEFINE}\ (f\ x_1\ \cdots\ x_k)\ e_f)$
$n : (\mathrm{OP}\ op\ n_1 : e_1\ n_2 : e_2)$	$\mathrm{Context}(n_1) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \circ$ $\mathrm{Context}(n_2) = \text{if } \mathrm{Context}(n) = \perp \text{ then } \perp \text{ else } \circ$
$(\mathrm{DEFINE}\ (main\ x_1\ \cdots\ x_k)\ n_0 : e_{main})$	$\mathrm{ContextEnv}(main) = \pi_{main}$ $\mathrm{ContextEnv}(x_i) = \bigsqcup_{m : x_i \in e_{main}} \mathrm{Context}(x_i)$ $\mathrm{Context}(n_0) = \mathrm{ContextEnv}(main)$
$(\mathrm{DEFINE}\ (f\ x_1\ \cdots\ x_k)\ n_0 : e_f)$	$\mathrm{ContextEnv}(f) = \bigsqcup_{m : (\mathrm{CALL}\ f\ a_1\ \cdots\ a_k) \in \mathrm{CallsTo}(f)} \mathrm{Context}(m)$ $\mathrm{ContextEnv}(x_i) = \bigsqcup_{m : x_i \in e_f} \mathrm{Context}(x_i)$ $\mathrm{Context}(n_0) = \mathrm{ContextEnv}(f)$

Figure 4. Equations for context analysis. The desired solution of these equations is the least-fixed-point solution.

in the program's expression tree, and variables of the form ContextEnv(m), where m is the name of a function or a formal parameter.[2] These variables take on right-hand sides of regular-tree-grammar productions as their values. For example, ContextEnv(f), ContextEnv(x_i), and Context(n) are "right-hand-side-valued" variables that correspond to a function named f, a parameter named x_i, and a subexpression labeled n, respectively. We then find the least solution of these equations in the (syntactic) domain of right-hand sides of regular-tree-grammar productions. The solution to the Context equations is then interpreted as a regular tree grammar whose productions are of the form

[2] We assume that all formal parameters have unique names (*e.g.*, by qualifying them with the name of the function to which they belong).

$n \rightarrow$ value of Context(n).

This grammar associates each nonterminal (*i.e.*, program point) n with a set of access paths

$L(n) = L$(value of Context(n)),

which in turn represents (in a finite way) a projection function for nonterminal (program point) n.

Example. We illustrate context analysis for the following functional version of the line_char_count program from Figure 2:

```
(DEFINE (main str)                    -- LineCharCount
    (CALL LineCharCountAux str ′0 ′0))
(DEFINE (LineCharCountAux str lc cc)
    (IF (NULL str)
      (CONS lc cc)
      (IF (EQUAL (CAR str) ′nl)
        (CALL LineCharCountAux (CDR str) (OP + lc ′1) (OP + cc ′1))
        (CALL LineCharCountAux (CDR str)  lc (OP + cc ′1)))))
```

The annotated version of the program is

(DEFINE (main str) -- LineCharCount
n_0: (CALL LineCharCountAux n_1: str n_2: ′0 n_3: ′0))

(DEFINE (LineCharCountAux str lc cc)
n_4: (IF n_5: (NULL n_6: str)
n_7: (CONS n_8: lc n_9: cc)
n_{10}: (IF n_{11}: (EQUAL n_{12}: (CAR n_{13}: str) n_{14}: ′nl)
n_{15}: (CALL LineCharCountAux n_{16}: (CDR n_{17}: str)
n_{18}: (OP + n_{19}: lc n_{20}: ′1)
n_{21}: (OP + n_{22}: cc n_{23}: ′1))
n_{24}: (CALL LineCharCountAux n_{25}: (CDR n_{26}: str)
n_{27}: lc
n_{28}: (OP + n_{29}: cc n_{30}: ′1)))))

Suppose we want to slice LineCharCount with respect to slicing criterion $\langle \top . \bot \rangle$. The value of π_{main} is $\langle \top . \bot \rangle$; the initial value of Context for all program points and of ContextEnv for all functions and parameters is $\bot$. The values for the Context and ContextEnv variables in the least-fixed-point solution of the equations are:

ContextEnv(main) = $\langle \top . \bot \rangle$
ContextEnv(main:str) = $\circ \mid \langle \top . \{ n_{16}, n_{25} \} \rangle$

ContextEnv(LineCharCountAux) = $\langle \top . \bot \rangle$
ContextEnv(LineCharCountAux:str) = $\circ \mid \langle \top . \{ n_{16}, n_{25} \} \rangle$
ContextEnv(LineCharCountAux:lc) = $\top$
ContextEnv(LineCharCountAux:cc) = $\bot$

Context(n_0) = $\langle \top . \bot \rangle$
Context(n_1) = $\circ \mid \langle \top . \{ n_{16}, n_{25} \} \rangle$
Context(n_2) = $\top$
Context(n_3) = $\bot$
Context(n_4) = $\langle \top . \bot \rangle$
Context(n_5) = $\circ \mid \langle \bot . \bot \rangle$
Context(n_6) = $\circ \mid \langle \bot . \bot \rangle$
Context(n_7) = $\langle \top . \bot \rangle$
Context(n_8) = $\top$
Context(n_9) = $\bot$
Context(n_{10}) = $\langle \top . \bot \rangle$
Context(n_{11}) = $\circ \mid \langle \bot . \bot \rangle$
Context(n_{12}) = $\top$
Context(n_{13}) = $\langle \{ n_{12} \} . \bot \rangle$
Context(n_{14}) = $\top$
Context(n_{15}) = $\langle \top . \bot \rangle$
Context(n_{16}) = $\circ \mid \langle \top . \{ n_{16}, n_{25} \} \rangle$
Context(n_{17}) = $\langle \bot . \{ n_{16} \} \rangle$
Context(n_{18}) = $\top$
Context(n_{19}) = $\circ$
Context(n_{20}) = $\circ$
Context(n_{21}) = $\bot$
Context(n_{22}) = $\bot$
Context(n_{23}) = $\bot$
Context(n_{24}) = $\langle \top . \bot \rangle$
Context(n_{25}) = $\circ \mid \langle \top . \{ n_{16}, n_{25} \} \rangle$
Context(n_{26}) = $\langle \bot . \{ n_{25} \} \rangle$
Context(n_{27}) = $\top$
Context(n_{28}) = $\bot$
Context(n_{29}) = $\bot$
Context(n_{30}) = $\bot$

These values agree with our intuition. Slicing criterion $\langle \top . \bot \rangle$ means: "The line count is of interest, but not the character count." As we would hope, the arithmetic expressions concerned with computing the line count (program points n_2, n_{18}, and n_{27}) are all associated with $\top$ (*i.e.*, "needed"), but the arithmetic expressions that compute the character count (n_3, n_{21}, and n_{28}) are all associated with $\bot$.

We can trace the flow of these context values through the program as follows: The call to LineCharCountAux in main causes the context π_{main} to be propagated to n_4, the body of LineCharCountAux. This context then passes through the IF expression at n_4 to the CONS expression at n_7. Here the context $\langle \top . \bot \rangle$ is split up, generating context $\top$ for variable lc and context $\bot$ for cc. Because lc is one of the formal parameters of LineCharCountAux, its context is collected and propagated to the appropriate expressions at all of LineCharCountAux's call sites, which causes expressions n_2, n_{18}, and n_{27} to have the context $\top$. □

3.3. Creating the Slice

For slicing, we also need to create a simplified version of the program (*i.e.*, the slice itself). We can actually identify two different goals for what we mean by "slicing a functional program", which we call Type I and Type II slices.

In Weiser's original definition of slicing for imperative programs, a slice is obtained from the original program by deleting zero or more statements [37, pp. 353]. Type I slicing is the analogue for functional programs of Weiser's slicing operation: subexpressions of the program, rather than statements, are deleted. A *Type I slice* prunes the program as follows: For every subexpression whose context is $\bot$, the result of evaluating the expression will not be used, as long as the client of the sliced program abides by the access "contract" given by π_{main}. Consequently, it is safe to replace every such subexpression by the expression ′?. In other words, as long as the client of the sliced program abides by the access "contract" given by π_{main}, the values that can be inspected will be the same as those generated by the original main program. The Type I slicing operation is shown in Figure 5.

Example. The final program that results from slicing LineCharCount is as follows:

SliceExp($n : e$) =
  if Context(n) = $\bot$ **then** (QUOTE ?)
  else case e **of**

$n : x_i$:	x_i
n : (QUOTE c):	(QUOTE $\pi_{\text{Context}(n)}(c)$)
n : (CAR $n_1 : e_1$):	(CAR SliceExp($n_1 : e_1$))
n : (CDR $n_1 : e_1$):	(CDR SliceExp($n_1 : e_1$))
n : (ATOM $n_1 : e_1$):	(ATOM SliceExp($n_1 : e_1$))
n : (NULL $n_1 : e_1$):	(NULL SliceExp($n_1 : e_1$))
n : (EQUAL $n_1 : e_1$ $n_2 : e_2$):	(EQUAL SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$))
n : (CONS $n_1 : e_1$ $n_2 : e_2$):	(CONS SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$))
n : (IF $n_1 : e_1$ $n_2 : e_2$ $n_3 : e_3$):	(IF SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$) SliceExp($n_3 : e_3$))
n : (CALL f $n_1 : e_1 \cdots n_k : e_k$):	(CALL f SliceExp($n_1 : e_1$) $\cdots$ SliceExp(n_k e_k))
n : (OP op $n_1 : e_1$ $n_2 : e_2$):	(OP op SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$))

  endcase fi

Figure 5. Type I slicing: Given the results of context analysis, function SliceExp is applied to each function body to create the sliced program.

```
(DEFINE (main str)
    (CALL LineCharCountAux str ′0 ′?))
(DEFINE (LineCharCountAux str lc cc)
    (IF (NULL str)
        (CONS lc ′?)
        (IF (EQUAL (CAR str) ′nl)
            (CALL LineCharCountAux (CDR str) (OP + lc ′1) ′?)
            (CALL LineCharCountAux (CDR str) lc ′?))))
```

In this program, all expressions associated solely with the computation of the character count have been replaced by ′?.

(A simple clean-up step can be used to remove formal parameter cc from LineCharCountAux and the corresponding actual parameters at the three calls on LineCharCountAux.) □

In SliceExp, the case for a subexpression n of the form (QUOTE c) is handled by applying a projection function $\pi_{\mathrm{Context}(n)}$ to c. This function is constructed from the value of Context(n) as follows:

(i) We first normalize the regular tree grammar so the each branch of each pair consists of a single symbol: $\top$, $\perp$, or a (single) nonterminal. Normalization of the grammar is carried out by the following method (which we will call *join-normalization*):

- A set of nonterminals $\{ N_1, N_2, \ldots, N_k \}$ is replaced by a new nonterminal N and a production for N is added to the grammar; the right-hand side of the new production is the join of the right-hand sides of the productions for N_1, N_2, $\cdots$, and N_k. This process is repeated until each branch of each pair on the right-hand side of a production (including the newly introduced productions) consists of a single symbol.
- During this process, a table is kept of which new nonterminals correspond to which sets of old nonterminals, and this table is consulted to reuse new nonterminals whenever possible. Because there is a finite number of such nonterminal sets, the process must terminate.

(ii) Given such a normalized grammar, $\pi_{\mathrm{Context}(n)}$ is defined as follows:

$$\pi_{\top} = \lambda t.t$$
$$\pi_{\perp} = \lambda t.?$$
$$\pi_{\bigcirc} = \lambda t.\text{if atom}(t) \text{ then } t \text{ else ? fi}$$
$$\pi_{\langle A.B \rangle} = \lambda t.\text{if atom}(t) \text{ then ? else cons}(\pi_A(t), \pi_B(t)) \text{ fi}$$
$$\pi_{\bigcirc \mid \langle A.B \rangle} = \lambda t.\text{if atom}(t) \text{ then } t \text{ else cons}(\pi_A(t), \pi_B(t)) \text{ fi}$$

(For the sake of uniformity, in these rules we assume that A and B stand for either $\top$, $\perp$, or a nonterminal symbol.)

A *Type II* slice differs from a Type I slice because it is allowed to introduce *additional* material into the sliced program. A Type II slice prunes out “extra” information that is found in the programs created by Type I slicing. The reason that such extra information exists is that the context analysis of Section 3.2 is a monovariant analysis. Because different portions of the result of a function may be needed at different call sites, with a Type I slice a function may return more information than is needed at a specific call site. In addition, more information may be present in a variable than is needed at all uses of that variable. For this reason, a sliced program generated by a Type I slice may occasionally return more information than is actually needed. This does not present a problem as long as all accesses are confined to the “contract” implicit in π_{main}. However, there may be times when we want the slice to be “stingy”; we want it to remove unneeded information when it arises. For this purpose, we define the method for Type II slicing shown in Figure 6. In places where it can detect that unneeded information might be introduced, the Type II slicing procedure inserts an explicit call to an appropriate projection function to trim down the return value.

SliceExp($n : e$) =
  if Context(n) = $\perp$ **then** (QUOTE ?)
  else case e **of**

$n : x_i$:	**if** Context(n) = ContextEnv(x_i) **then** x_i **else** (CALL $\pi_{\text{Context}}(n)$ x_i) **fi**
n : (QUOTE c):	(QUOTE $\pi_{\text{Context}(n)}(c)$)
n : (CAR $n_1 : e_1$):	(CAR SliceExp($n_1 : e_1$))
n : (CDR $n_1 : e_1$):	(CDR SliceExp($n_1 : e_1$))
n : (ATOM $n_1 : e_1$):	(ATOM SliceExp($n_1 : e_1$))
n : (NULL $n_1 : e_1$):	(NULL SliceExp($n_1 : e_1$))
n : (EQUAL $n_1 : e_1$ $n_2 : e_2$):	(EQUAL SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$))
n : (CONS $n_1 : e_1$ $n_2 : e_2$):	(CONS SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$))
n : (IF $n_1 : e_1$ $n_2 : e_2$ $n_3 : e_3$):	(IF SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$) SliceExp($n_3 : e_3$))
n : (CALL f $n_1 : e_1 \cdots n_k : e_k$):	**if** Context(n) = ContextEnv(f) **then** (CALL f SliceExp($n_1 : e_1$) $\cdots$ SliceExp(n_k e_k)) **else** (CALL $\pi_{\text{Context}}(n)$ (CALL f SliceExp($n_1 : e_1$) $\cdots$ SliceExp(n_k e_k))) **fi**
n : (OP op $n_1 : e_1$ $n_2 : e_2$):	(OP op SliceExp($n_1 : e_1$) SliceExp($n_2 : e_2$))

  endcase fi

Figure 6. Type II slicing: A second method for slicing a functional program. Because of the monovariant treatment of functions, variables and function calls may carry "too much" information for their actual context. To remedy this, calls to an appropriate projection function are generated.

(In the LineCharCount example, the Type I and Type II slices are identical; no projection functions would be inserted by the Type II slicing method.)

3.4. An Improved Slicing Algorithm

A further improvement of the slicing algorithm can be obtained by combining *shape information* with context information. To describe this extension, it is convenient to give a formulation of the shape-analysis problem in a way that is similar in style to the context-analysis equation schemas of Figure 4. The equation schemas for shape analysis are presented in Figure 7. In shape analysis, regular tree grammars are used as shape descriptors to summarize the possible shapes of values (as characterized by a set of access paths) that may be returned by a subexpression.

To be able to combine shape information with context information, we also need the operation on right-hand sides of regular-tree-grammar productions that is defined as follows:

$$M \oplus N = \text{true iff} \begin{cases} \quad M = \perp \\ \text{or} \quad N = \perp \\ \text{or} \quad M = \circ \text{ and } N = \langle A.B \rangle \\ \text{or} \quad N = \circ \text{ and } M = \langle A.B \rangle \end{cases}$$

The improvement to the slicing algorithm consists of replacing the first line of (either version of) SliceExp with

if Shape(n) $\oplus$ Context(n) = true **then** (QUOTE ?)

The reason this is safe is that if Shape(n) $\oplus$ Context(n) = true at subexpression n, then the value created at

Form of expression	Equations associated with expression
$n : x_i$	Shape(n) = ShapeEnv(x_i)
n : (QUOTE c)	Shape(n) = ConstShape(c)
n : (CAR $n_1 : e_1$)	$\text{Shape}(n) = \begin{cases} \top & \text{if Shape}(n_1) = \top \text{ or } \circ \mid \langle \top . B \rangle \text{ or } \langle \top . B \rangle \\ \bot & \text{if Shape}(n_1) = \bot \text{ or } \circ \text{ or } \circ \mid \langle \bot . B \rangle \text{ or } \langle \bot . B \rangle \\ \bigsqcup_{A_i \in A} A_i & \text{if Shape}(n_1) = \circ \mid \langle A . B \rangle \text{ or } \langle A . B \rangle \end{cases}$
n : (CDR $n_1 : e_1$)	$\text{Shape}(n) = \begin{cases} \top & \text{if Shape}(n_1) = \top \text{ or } \circ \mid \langle B . \top \rangle \text{ or } \langle B . \top \rangle \\ \bot & \text{if Shape}(n_1) = \bot \text{ or } \circ \text{ or } \circ \mid \langle B . \bot \rangle \text{ or } \langle B . \bot \rangle \\ \bigsqcup_{A_i \in A} A_i & \text{if Shape}(n_1) = \circ \mid \langle B . A \rangle \text{ or } \langle B . A \rangle \end{cases}$
n : (ATOM $n_1 : e_1$)	Shape(n) = if Shape(n_1) = $\bot$ then $\bot$ else $\circ$
n : (NULL $n_1 : e_1$)	Shape(n) = if Shape(n_1) = $\bot$ then $\bot$ else $\circ$
n : (EQUAL $n_1 : e_1$ $n_2 : e_2$)	Shape(n) = if Shape(n_1) = $\bot$ then $\bot$ else $\circ$
n : (CONS $n_1 : e_1$ $n_2 : e_2$)	Shape(n) = ({ n_1 } . { n_2 })
n : (IF $n_1 : e_1$ $n_2 : e_2$ $n_3 : e_3$)	$\text{Shape}(n) = \begin{cases} \bot & \text{if Shape}(n_1) = \bot \\ \text{Shape}(n_2) \sqcup \text{Shape}(n_3) & \text{otherwise} \end{cases}$
n : (CALL $f\ e_1 \cdots e_k$)	Shape(n) = ShapeEnv(f)
n : (OP op $n_1 : e_1$ $n_2 : e_2$)	Shape(n) = if (Shape(n_1) = $\circ$) and (Shape(n_2) = $\circ$) then $\circ$ else $\bot$
(DEFINE ($main\ x_1 \cdots x_k$) $n_0 : e_{main}$)	ShapeEnv($main$) = Shape(n_0) ShapeEnv(x_i) = InitialShapeEnv(x_i)
(DEFINE ($f\ x_1 \cdots x_k$) $n_0 : e_f$)	ShapeEnv(f) = Shape(n_0) $\text{ShapeEnv}(x_i) = \bigsqcup_{m : (\text{CALL } f\ n_1 : a_1 \cdots n_k : a_k) \in \text{CallsTo}(f)} \text{Shape}(n_i)$

Figure 7. Equations for shape analysis. The desired solution of these equations is the least-fixed-point solution. Auxiliary function ConstShape(c) returns a shape descriptor for a constant c. InitialShapeEnv is a map from *main*'s formal parameters to their (known) initial shape descriptors.

n can never contain any of the access paths in L(Context(n)). Because we are limiting attention to the portions of n's possible return values that lie on the access paths in L(Context(n))—of which there are none—we can replace subexpression n with ′?.

Example. Suppose we use slicing criterion $\circ$ to slice the following program:

```
(DEFINE (main) (CALL mycons ′1 ′2))
(DEFINE (mycons x y) (CONS x y)).
```

Without the suggested improvement, both versions of SliceExp create the program

```
(DEFINE (main) (CALL mycons ′? ′?))
(DEFINE (mycons x y) (CONS ′? ′?)),
```

which contains a wasted function call and also returns a value that contains extra information. With the improvement, both versions of SliceExp create

```
(DEFINE (main) ′?)
(DEFINE (mycons x y) ′?).
```

□

4. Semantic Issues

We do not have space in this paper for an in-depth treatment of the issue of how the semantics of a slice relates to the semantics of the original program. In fact, our techniques do not guarantee that equation (†) of Section 3.2 holds (*i.e.*, $[\![q]\!] = \pi_{main} \circ [\![p]\!]$) unless the programming language has a lazy semantics. The reason is that, for a call-by-value language, slicing may change the termination behavior; that is, a slice may terminate on inputs on which the original program diverges. Slicing can never introduce *divergence*; it can only introduce *termination*, which, from a pragmatic standpoint, is a quite reasonable situation.

Example. For (single-procedure) programs in imperative languages, the need for a lazy semantics can be illustrated by means of the following example: Consider the three programs P_1, P_2, and P_3:

P_1	P_2	P_3
x = 0;	x = 0;	x = 0;
for (i = 1; ; i++) { }	w = 1;	$\boxed{}$
y = x;	y = x;	y = x;

P_3 is the slice of P_1 with respect to y = x; P_3 is also the slice of P_2 with respect to y = x. Let $\sqsubseteq_{sl}$ denote the “is-a-slice-of” relation, and let $\sqsubseteq_{sem}$ denote the semantic approximation relation.

In a standard direct denotational semantics for an imperative language (denoted by $\mathbf{M}[\![\cdot]\!]$), commands are (strict) store-to-store transformers. Because program P_1 contains an infinite loop, we have $\mathbf{M}[\![P_1]\!] = \lambda s.\bot_{store}$. Consequently, even though

$$P_3 \sqsubseteq_{sl} P_1$$

and

$$P_3 \sqsubseteq_{sl} P_2,$$

we have

$$\mathbf{M}[\![P_1]\!] \sqsubseteq_{sem} \mathbf{M}[\![P_3]\!] \sqsubseteq_{sem} \mathbf{M}[\![P_2]\!].$$

In other words, with the standard treatment of the semantics of imperative languages, the relation “is-a-slice-of” is not consistent with the semantic approximation relation.

However, there is a non-standard setting in which the hoped-for relationships *do* hold. Ramalingam and Reps have defined an equational *value-sequence*-oriented semantics (as opposed to a conventional *state-oriented* semantics) for a variant of the program dependence graph [26] (see also [5]). Rather than treating each program point as a state-to-state transformer, the value-sequence semantics treats each program point as a *value-sequence transformer* that takes (possibly infinite) argument sequences from dependence predecessors to a (possibly infinite) output sequence. The latter sequence represents the *sequence of values* computed at that point during program execution. Because dependence edges can bypass infinite loops, the value-sequence semantics is *more defined than* a standard operational or denotational semantics. For example, the vertex for statement y = x in program P_1 has the singleton sequence “[0]” rather than, for example, the uncompleted sequence “$\bot$”. (This agrees with the sequence for y = x in program P_3, which is also “[0]”.)

With the value-sequence semantics, it is trivial to show that $\sqsubseteq_{sl}$ and $\sqsubseteq_{sem}$ are consistent: A slice is the subgraph of the dependence graph found by following edges of the dependence graph backwards from the vertex v of interest; this subgraph corresponds exactly to the subset S_v of the equations that can affect the value-sequence at v. Because we followed *all* paths backwards from v to identify S_v, the solutions to equation system S_v and to the full equation system must be identical for all vertices that occur in both the program and the slice. Consequently, in this framework the semantics of a slice approximates the semantics of the full program (and never vice versa).

(Other approaches to lazy semantics for program dependence graphs include [30], [4], and [1].) □

For readers who are uncomfortable with the "semantic anomaly" that a slice does not preserve the termination behavior of the original program, we would like to point out that this situation is far more acceptable than the semantic anomaly exhibited by partial evaluation, where, due to the well-known problems with non-termination of partial evaluators in the presence of static-infinite computations ([16, pp. 265-266], [31, pp. 501-502], [22, pp. 337], [17, pp. 299], and [5]), partial evaluation can introduce *divergence*. That is, the specializer *itself* can diverge on programs that would not diverge on all dynamic inputs if executed in their unspecialized form. In other words,

- Partial evaluation is faithful to the termination properties of the original program only under the assumption that programs in the language are "*hyper-strict*".
- Similarly, but with less potential for disruptive behavior, program slicing is faithful to the termination properties of the original program only under the assumption that the language has *lazy semantics*.

While no reasonable programming languages have hyper-strict semantics, there do exist programming languages with lazy semantics.

5. A Re-Examination of Program Bifurcation

In [23], Mogensen describes a method to perform program bifurcation. Briefly stated, bifurcation is a way to transform a program that takes partially static structures as arguments into two programs: one in which all of the parameters are totally static, and a second in which all of the parameters are either totally static or totally dynamic. This section outlines how some of the steps used in program bifurcation can be redefined in terms of program slicing.

Mogensen defines operations that split a function f into a function f_S, which computes the purely static part of f's result, and a function f_D, which computes the dynamic part of f's result. We will refer to these operations as BifS and BifD, respectively. BifS identifies and removes all possibly dynamic expressions; BifD identifies and removes static expressions whose values are not needed for the dynamic result. In Mogensen's formulation of them, BifS and BifD do not incorporate a true neededness analysis; instead, they use binding-time information (which is computed by propagating information through the program in the forward direction) as a sort of "pseudo-neededness" information.

Mogensen begins by performing binding-time analysis. He uses a domain of regular tree grammars that is much like the domain we use: his symbol S corresponds to our symbol $\top$; his D, to our $\perp$; and his *atomS*, to our $\circ$. The binding-time analysis can be defined as the fixed point of a set of equations that are similar to our shape-analysis and context-analysis equations. (We have omitted this reformulation for reasons of space.) The binding-time analysis produces, for pertinent expressions in the program, a regular tree grammar describing "how static" each expression is guaranteed to be. Specifically, when interpreted as a prefix-closed set of strings, the regular-tree-grammar production associated with a subexpression n describes (a subset of) the set of all access paths that are guaranteed not to lead to data that is dynamic in any value returned by n. (This is not to say that all access paths described by the grammar rule for n necessarily exist in each of n's possible return values.) The final step of bifurcation is to apply BifS and BifD to each function of the program.

BifS and BifD are similar to, but not precisely, Type II slices. This motivates us to formulate revised bifurcation operations, called BifS′ and BifD′, that are based on program slicing (and hence *do* perform a true neededness analysis). Some of the advantages of using a "true" over a "pseudo" neededness analysis are as follows:

- For BifS′, static expressions whose values are not needed for the static result can now be identified and removed.

- For BifD′, dead dynamic code can now be identified and removed.

Below, we will only illustrate bifurcation procedure BifS′. (Because BifD′ uses BifS′ as a subroutine, some, but not all, of the differences between BifD′ and BifD are inherited from the differences between BifS′ and BifS.) The BifS′ procedure is as follows:

(i) Binding-time analysis is performed, using the given (possibly partially static) binding times for *main*'s variables.
(ii) A "meet-normalization" procedure is applied to the resulting regular tree grammar. (The grammars used in Mogensen's binding-time analysis are of a kind that is dual to the kind we use, and hence the results of binding-time analysis must be normalized by a process dual to the join-normalization operation defined in Section 3.3. This converts the productions of the grammar obtained from binding-time analysis to ones in which each branch of each right-hand-side pair is a singleton set. The normalized grammar can then be interpreted as a grammar of the kind we are using for slicing.)
(iii) The nonterminals in the normalized grammar are systematically renamed to remove all uses of the names of the program's functions, formal parameters, and subexpressions.
(iv) A Type II slice of the program is then performed, using the (renamed) binding-time descriptor for function *main* as the slicing criterion π_{main}.

Example. Consider the program:

```
(DEFINE (main a b c d)
        (IF c (CONS a b) (CONS a d)))
```

with the input division[3]

a: S, b: S, c: S, d: D.

The value of π_{main} generated by the binding-time analysis is $\langle S.D \rangle$. The program generated by BifS is

(DEFINE (main a b c d)
        (IF c (CALL $\pi_{\langle S.D \rangle}$ (CONS a b)) (CONS a ′?))),

whereas BifS′ yields

```
(DEFINE (main a b c d)
        (IF c (CONS a ′?) (CONS a ′?))))
```

The expression b is retained by BifS because it is static, whereas BifS′ classifies the expression as unneeded ($\perp$) and prunes it from the sliced program. □

While the differences between what the two versions of BifS produce are not earthshaking, they provide another viewpoint for understanding the results presented in this paper:

- The essence of the rewriting step used in program bifurcation is the analogue for functional programs of program slicing (which had been defined earlier for imperative programs).
- Slicing of functional programs is a program-specialization operation of interest in its own right and can be isolated from the rest of the machinery that is part of bifurcation.

6. Relation to Previous Work

In the slicing community, slicing has long been recognized as a way to specialize programs. Many of the proposed applications of slicing are based on its properties as a specialization operation. For example,

---

[3]In this example, we are using a division in which none of the parameters are partially static. This is done merely to give the simplest possible example of the differences between BifS and BifS′.

- Weiser proposed using slicing to decompose programs into separate threads that can be run in parallel [36]. Each thread computes a portion of what is computed by the original program.
- Horwitz, Reps, and Prins proposed an algorithm for merging two variants *A* and *B* of a program *Base* [11]. The algorithm breaks down *Base*, *A*, and *B* into their constituent slices and chooses among them to create the merged program. By selecting appropriate slices, the algorithm guarantees that the merged program exhibits all changed behavior of *A* with respect to *Base*, all changed behavior of *B* with respect to *Base*, and all behavior that is common to all three [27].
- Bates and Horwitz proposed to use slicing to avoid redoing the part of a test suite that is unaffected by a change to a program [2].
- Andersen Consulting's interactive Cobol System Renovation Environment (Cobol/SRE) is a system for re-engineering legacy systems written in Cobol [24]. It uses slicing as the fundamental operation that users employ to select program fragments of interest. Operations are provided for combining slices (*i.e.*, union, intersection, and difference). These fragments are then used to reorganize the program by extracting the code fragments and repackaging them into independent modules.

Most work on slicing has concerned imperative programming languages. In the context of functional languages, a slicing-like operation is used by Liu and Teitelbaum as a cleanup step in their transformational methodology for deriving incremental versions of functional programs from non-incremental functional programs [20]. In their work, slices can be taken only with respect to projection functions that express finite-depth access patterns in a tree. In contrast, the method we have presented uses regular-tree grammars to express projection functions that have arbitrary-depth (but regular) access patterns.

This paper concerns the complementary relationship between slicing and partial evaluation when *backward* slicing is considered as a specialization operation. Another kind of relationship between slicing and partial evaluation has been established by Das, Reps, and Van Hentenryck who showed how three variants of *forward* slicing can be used to carry out binding-time analysis for imperative programs [5].

This paper has been greatly influenced by the literature on partial evaluation and related operations, particularly by Mogensen's paper on program bifurcation [23]. In particular, the variant of regular tree grammars that we have used is based on Mogensen's work (as opposed to the version of regular tree grammars used by Jones and Muchnick [15] and the normalized set equations used by Reynolds [29]).

The context analysis that we have used to define the slicing algorithm is related to the "neededness" analysis defined by Hughes [14] and also to the "strictness analysis" of Wadler and Hughes, which is also capable of identifying whether the value of a subexpression is ignored [34, pp. 392]. Our analysis is somewhat different from these two and, in general, incomparable to them. For instance, the latter analyses are both formulated in terms of a fixed, finite set of projection functions for characterizing "neededness patterns" of list-manipulation programs. The use of a fixed set of projection functions makes the analysis efficient, but it also introduces some limitations on the class of neededness patterns that can be identified. (This is not to say that only uninteresting neededness information can be discovered. On the contrary, Hughes's analysis is able to determine that in the length function the spine of the argument list is needed, but the elements of the list are not needed.) Because our work is based on regular tree grammars, our analysis is capable of handling a broader class of neededness patterns: The advantage of the regular-tree grammar approach is that it "adapts" to the patterns that are used in a particular program.

Another issue concerns the monovariant versus polyvariant treatment of functions. In the work of Hughes, Wadler and Hughes, and Liu and Teitelbaum, the context analyses that are used create projection-function transformers for each source-program function, which are then employed at each call site to determine how the call site's local context is transformed. This is only feasible when the domain of projection functions is small (*e.g.* Wadler and Hughes work with a 10-point domain of projection functions). Because our domain of projection functions—regular-tree grammars—is large, our work follows Mogensen and uses a monovariant analysis (*i.e.*, the contexts of all calls to a function *f* are combined to determine the context of *f*'s body). This monovariant analysis loses precision, but the alternative polyvariant analysis would involve tabulating a collection of functions of type "regular-tree-grammar → regular-tree-grammar".

7. Concluding Remarks

This paper has shown how program slicing can be used to carry out a certain kind of program-specialization operation. Because the paper extends existing slicing techniques by making use of techniques that are closely related to ones that have been used in both partial evaluation and program bifurcation, the paper serves to bridge the gap between two communities—the partial-evaluation community and the program-slicing community—that are both working on semantics-based program manipulation but that (to date) have had relatively limited contact. For these two communities, the salient connections to the material presented in the paper are as follows:

- Our results should be of interest to the partial-evaluation community because we have demonstrated a new way of specializing programs that is different from the specialization operations carried out by partial evaluation, supercompilation, bifurcation, and deforestation. In addition, the slicing-based specialization operation has another characteristic that sets it apart from partial evaluation (and other forwards-oriented specialization operations): Slicing-based specialization permits programs to be specialized in ways that do not have to be anticipated by the author of the original program (in the sense that specialization is not linked to the parameters to functions and procedures provided in the original program).
- Our results should also be of interest to the program-slicing community because the techniques presented in the paper go beyond existing slicing techniques. We present a setting in which it is possible to carry out non-trivial slices of programs that make use of heap-allocated data structures; in this setting, programs are sliced with respect to "partially needed structures".

References

1. Ballance, R.A., Maccabe, A.B., and Ottenstein, K.J., "The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 257-271 (June 1990).
2. Bates, S. and Horwitz, S., "Incremental program testing using program dependence graphs," pp. 384-396 in *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages,* (Charleston, SC, January 10-13, 1993), ACM, New York, NY (1993).
3. Binkley, D., "Using semantic differencing to reduce the cost of regression testing," *Proceedings of the 1992 Conference on Software Maintenance* (Orlando, FL, November 9-12, 1992), pp. 41-50 (1992).
4. Cartwright, R. and Felleisen, M., "The semantics of program dependence," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 13-27 (July 1989).
5. Das, M., Reps, T., and Van Hentenryck, P., "Semantic foundations of binding-time analysis for imperative programs," pp. 100-110 in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 95),* (La Jolla, California, June 21-23, 1995), ACM, New York, NY (1995).
6. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.* **9**(3) pp. 319-349 (July 1987).
7. Futamura, Y. and Nogi, K., "Generalized partial computation," pp. 133-152 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation,* (Gammel Avernaes, Denmark, October 18-24, 1987), ed. D. Bjϕrner, A.P. Ershov, and N.D. Jones,North-Holland, New York, NY (1988).
8. Gallagher, K.B., "Using program slicing in software maintenance," Ph.D. dissertation and Tech. Rep. CS-90-05, Computer Science Department, University of Maryland, Baltimore Campus, Baltimore, MD (January 1990).
9. Gallagher, K.B. and Lyle, J.R., "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering* **SE-17**(8) pp. 751-761 (August 1991).
10. Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices*

24(7) pp. 28-40 (July 1989).

11. Horwitz, S., Prins, J., and Reps, T., “Integrating non-interfering versions of programs,” *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).
12. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
13. Horwitz, S., “Identifying the semantic and textual differences between two versions of a program,” *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 234-245 (June 1990).
14. Hughes, J., “Backwards analysis of functional programs,” pp. 187-208 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation,* (Gammel Avernaes, Denmark, October 18-24, 1987), ed. D. Bjϕrner, A.P. Ershov, and N.D. Jones,North-Holland, New York, NY (1988).
15. Jones, N.D. and Muchnick, S.S., “Flow analysis and optimization of Lisp-like structures,” pp. 102-131 in *Program Flow Analysis: Theory and Applications,* ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).
16. Jones, N.D., “Automatic program specialization: A reexamination from basic principles,” pp. 225-282 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation,* (Gammel Avernaes, Denmark, October 18-24, 1987), ed. D. Bjϕrner, A.P. Ershov, and N.D. Jones,North-Holland, New York, NY (1988).
17. Jones, N.D., Gomard, C.K., and Sestoft, P., *Partial Evaluation and Automatic Program Generation,* Prentice-Hall International, Englewood Cliffs, NJ (1993).
18. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., “Dependence graphs and compiler optimizations,” pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).
19. Launchbury, J., *Projection Factorizations in Partial Evaluation*, Cambridge University Press, Cambridge, UK (1991).
20. Liu, Y.A. and Teitelbaum, T., “Caching intermediate results for program improvement,” in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 95),* (La Jolla, California, June 21-23, 1995), ACM, New York, NY (1995).
21. Lyle, J. and Weiser, M., “Experiments on slicing-based debugging tools,” in *Proceedings of the First Conference on Empirical Studies of Programming,* (June 1986), Ablex Publishing Co. (1986).
22. Mogensen, T., “Partially static structures in a self-applicable partial evaluator,” pp. 325-347 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation,* (Gammel Avernaes, Denmark, October 18-24, 1987), ed. D. Bjϕrner, A.P. Ershov, and N.D. Jones,North-Holland, New York, NY (1988).
23. Mogensen, T., “Separating binding times in language specifications,” pp. 12-25 in *Fourth International Conference on Functional Programming and Computer Architecture,* (London, UK, Sept. 11-13, 1989), ACM Press, New York, NY (1989).
24. Ning, J.Q., Engberts, A., and Kozaczynski, W., “Automated support for legacy code understanding,” *Commun. of the ACM* **37**(5) pp. 50-57 (May 1994).
25. Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).
26. Ramalingam, G. and Reps, T., “Semantics of program representation graphs,” TR-900, Computer Sciences Department, University of Wisconsin, Madison, WI (December 1989).
27. Reps, T. and Yang, W., “The semantics of program slicing and program integration,” pp. 360-374 in *Proceedings of the Colloquium on Current Issues in Programming Languages,* (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science,* Vol. 352, Springer-Verlag, New York, NY (1989).
28. Reps, T., “Shape analysis as a generalized path problem,” pp. 1-11 in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 95),* (La Jolla, California, June 21-23, 1995), ACM, New York, NY (1995).
29. Reynolds, J.C., “Automatic computation of data set definitions,” pp. 456-461 in *Information Processing 68: Proceedings of the IFIP Congress 68*, North-Holland, New York, NY (1968).
30. Selke, R.P, “A rewriting semantics for program dependence graphs,” pp. 12-24 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
31. Sestoft, P., “Automatic call unfolding in a partial evaluator,” pp. 485-506 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation,* (Gammel Avernaes, Denmark, October 18-24, 1987), ed. D. Bjϕrner, A.P. Ershov, and N.D. Jones,North-Holland, New York, NY (1988).
32. Tip, F., “A survey of program slicing techniques,” *Journal of Programming Languages* **3** pp. 121-181 (1995).
33. Turchin, V.F., “The concept of a supercompiler,” *ACM Trans. Program. Lang. Syst.* **8**(3) pp. 292-325 (July 1986).
34. Wadler, P. and Hughes, R.J.M., “Projections for strictness analysis,” pp. 385-407 in *Third Conference on Functional Programming and Computer Architecture,* (Portland, OR, Sept. 14-16, 1987), *Lecture Notes in Computer Science,* Vol. 274, ed. G. Kahn,Springer-Verlag, New York, NY (1987).
35. Wadler, P., “Deforestation: Transforming programs to eliminate trees,” *Theoretical Computer Science* **73** pp. 231-248 (1990).
36. Weiser, M., “Reconstructing sequential behavior from parallel behavior projections,” *Information Processing Letters* **17** pp. 129-135 (October 1983).

37. Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).