Interprocedural Control Flow Analysis of First-Order Programs with Tail Call Optimization*

Saumya K. Debray and Todd A. Proebsting
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA.
Email: {debray, todd}@cs.arizona.edu

Technical Report 96-20

December 2, 1996

Abstract

The analysis of control flow
Involves figuring out where `returns` will go.
How this may be done
With items LR-0 and -1
Is what in this paper we show.

*The work of S. Debray was supported in part by the National Science Foundation under grant number CCR-9502826. The work of T. Proebsting was supported in part by NSF Grants CCR-9415932, CCR-9502397, ARPA Grant DABT63-95-C-0075, and IBM Corporation.

1 Introduction

Most code optimizations depend on control flow analysis, typically expressed in the form of a control flow graph [1]. Traditional algorithms construct intraprocedural flow graphs, which do not account for control flow between procedures. Optimizations that depend on this limited information cannot consider the behavior of other procedures. Interprocedural versions of these optimizations must capture the flow of control across procedure boundaries. Determining interprocedural control flow (for first-order programs) is relatively straightforward in the absence of tail call optimization, since procedures return control to the point immediately after the call. Tail call optimization complicates the analysis because returns may transfer control to a procedure other than the active procedure's caller.

The problem can be illustrated by the following simple program that takes a list of values and prints, in their original order, all the values that satisfy some property, e.g., exceed 100. To take advantage of tail-call optimization, it uses an accumulator to collect these values as it traverses the input list. However, this causes the order of the values in the accumulator to be reversed, so the accumulated list is reversed—again using an accumulator—before it is returned.

```
(1)   main(L) = print extract(L, [])

(2)   extract(xs, acc) =
(3)      if xs = [] then reverse(acc, [])
(4)      else if hd(xs) > 100 then extract(tl(xs), cons(hd(xs),acc))
(5)      else extract(tl(xs), acc)

(6)   reverse(xs, acc) =
(7)      if xs = [] then acc
(8)      else reverse(tl(xs), cons(hd(xs),acc))
```

Suppose that, for code optimization purposes, we want to construct a flow graph for this entire program. The return from the function `reverse` in line 7 corresponds to some basic block, and in order to construct the flow graph we need to determine the successors of this block. The call graph of the program indicates that `reverse` can be called either from `extract`, in line 3, or from `reverse`, in line 8. However, because of tail call optimization, it turns out that `reverse` does not return to either of these call sites. Instead, it returns to a different procedure entirely, namely, to the procedure `main` in line 1: the successor block to the return in line 7 is the basic block that calls `print`. Clearly, some nontrivial control flow analysis is necessary to determine this.

Most of the work to date on control flow analysis has focused on higher-order languages: Shivers [17, 18] and Jagannathan and Weeks [7] use abstract interpretation for this purpose, while Heintze [4] and Tang and Jouvelot [19, 20] use type-based analyses. These analyses are very general, but very complex. Many widely used languages, such as Sisal and Prolog, are first-order languages. Furthermore, even for higher-order languages, specific programs often use only first-order constructs, or can have most higher-order constructs removed via transformations such as inlining and uncurrying [21]. As a pragmatic issue, therefore, we are interested in "ordinary" first-order programs: our aim is to account for interprocedural control flow in such programs in the presence of tail call optimization. To our knowledge, the only other work addressing this issue is that of Lindgren [9], who uses set-based analysis for control flow analysis of Prolog. Unlike Lindgren's work, our analyses can maintain context information (see Section 6).

The main contribution of this paper is to show how control flow analysis of first-order programs with tail call optimization can be formulated in terms of simple and well-understood concepts from parsing theory. In particular, we show that context-insensitive, or zeroth-order, control flow analysis corresponds to the notion of FOLLOW sets in context free grammars, while context-sensitive, or first-order, control flow analysis corresponds to the notion of LR(1) items. This is useful, because it allows the immediate application of well-understood technology without, for example, having to construct complex abstract domains. It is also esthetically pleasing, in that it provides an application of concepts such as FOLLOW sets and LR(1) items, which were originally developed purely in the context of parsing, to a very different application.

The remainder of the paper is organized as follows. Section 2 introduces definitions and notation. Section 3 defines an abstract model for control flow, and Section 4 shows how this model can be described using context free grammars. Section 5 discusses control flow analysis that maintain no context information, and Section 6 discusses how context information can be maintained to produce more precise analyses. Section 7 illustrates these ideas with a nontrivial example. Section 8 discusses tradeoffs between efficiency and precision.

In order to maintain continuity, the proofs of theorems have been relegated to the appendix.

2 Definitions and Notation

We assume that a program consists of a set of procedure definitions, together with an entry point procedure. (It is straightforward to extend these ideas to accommodate multiple entry points.) Since we assume a first-order language, the intraprocedural control flow can be modelled by a control flow graph [1]. This is a directed graph where each node corresponds to a basic block, i.e., a (maximal) sequence of executable code that has a single entry point and a single exit point, and where there is an edge from a node A to a node B if and only if it is possible for execution to leave node A and immediately enter node B. If there is an edge from a node A to a node B, then A is said to be a *predecessor* of B and B is a *successor* of A. Because of the effects of tail call optimization, interprocedural control flow information cannot be assumed to be available. Therefore, we assume that the input to our analysis consists of one control flow graph for each procedure defined in the program.

For simplicity of exposition, we assume that each flow graph has a single entry node. Each flow graph consists of a set of vertices, which correspond to basic blocks, and a set of edges, which capture control flow between basic blocks. If a basic block contains a procedure call, the call is assumed to terminate the block; if a basic block B ends in a call to a procedure p, we say that B *calls* p.

If the last action along an execution path in a procedure p is a call to some procedure q—i.e., if the only action that would be performed on return from q would be to return to the caller of p—the call to q is termed a *tail call*. A tail call can be optimized: in particular, any environment allocated for the caller p can be deallocated, and control transfer effected via a direct jump to the callee q; this is usually referred to as "tail call optimization," and is crucial for efficient implementations of functional and logic languages. If a basic block B ends in a tail call, we say that it is a *tail call block*; if B ends in a procedure call that is not a tail call, we say B is a *call block*. In the latter case, B must set a return address L before making the call: L is said to be a *return label*. If a basic block B ends in a return from a procedure, it is said to be a *return block*. As is standard in the program analysis literature, we assume that either branch of a conditional can be executed at runtime. The ideas described here are applicable to programs that do not satisfy this assumption; in that case, the analysis results will be sound but possibly conservative.

The set of basic blocks and labels appearing in a program P are denoted by Blocks_P and Labels_P respectively. The set of procedures defined in it is denoted by Procs_P. Finally, the Kleene closure of a set S, i.e., the set of all finite sequences of elements of S, is written S^*. The reflexive transitive closure of a relation R is written $R^\star$.

3 Abstracting Control Flow

Before we can analyze the control flow behavior of such programs, it is necessary to specify this behavior carefully. First, consider the actual runtime behavior of a program:

- Code not involving procedure calls or returns is executed as expected: each instruction in a basic block is executed in turn, after which control moves to a successor block, and so on.
- Procedure calls are executed as follows.
  - A *non-tail call* loads arguments into the appropriate locations, saves the return address (for simplicity, we can assume that it is pushed on a control stack), and branches to the callee.
  - A *tail call* loads arguments, reclaims any space allocated for the caller's environment, and transfers control to the callee.

– A procedure return simply pops the topmost return address from the control stack and transfers control to this address.

We can ignore any aspect of a program's runtime behavior that is not concerned directly with flow of control. Conceptually, therefore, control moves from one basic block to another, pushing a return address on a stack when making a non-tail procedure call, and popping an address from it when returning from a procedure. This can be formalized using a very simple pushdown automaton: the automaton M_P corresponding to a program P is called its *control flow automaton*. Given a program P, the set of states Q of M_P is given by $Q = \mathsf{Blocks}_P \cup \mathsf{Procs}_P$; its input alphabet is Labels_P; the initial state of M_P is p, where p is the entry point of P; and its stack alphabet $\Gamma = \mathsf{Labels}_P \cup \mathsf{Blocks}_P \cup \{\$\}$, where \$ is a special bottom-of-stack marker that is the initial stack symbol.

The general idea is that the state of M_P at any point corresponds to the basic block being executed by P, while the return labels on its stack correspond to the stack of procedure calls in P. The input string does not play a direct role in determining the behavior of M_P, but it turns out to be technically very convenient to match up symbols read from the input with labels popped from the stack. The language accepted by M_P is then the set of sequences of labels that control can jump to on procedure returns during an execution of the program P.

Let the transitions of a pushdown automaton be denoted as follows [6]: if, from a configuration where it is in state q and has w in its input and α on its stack, it can make a transition to state q' with input string w' and with β on its stack, we write $(q, w, \alpha) \vdash (q', w', \beta)$. The stack contents β are written such that the top of the stack is to the left: if $\beta \equiv a_1 \dots a_n$ then a_1 is assumed to be at the top of the stack. The moves of M_P are defined as follows:

1. If basic block B is a predecessor of basic block B', and B does not make a call or tail call, then M_P can make an ε-move from B to B':

   $$(B, w, \alpha) \vdash (B', w, \alpha)$$

2. If basic block B makes a call to procedure p with return label ℓ, where the basic block with label ℓ is B', then M_P can push two symbols $\ell B'$ on its stack and make an ε-move to state p:

   $$(B, w, \alpha) \vdash (p, w, \ell B' \alpha)$$

3. If basic block B makes a tail call to procedure p, then M_P can make an ε-move to state p:

   $$(B, w, \alpha) \vdash (p, w, \alpha)$$

4. If the entry node of the flow graph of a procedure p is B, then M_P can make an ε-move from state p to state B:

   $$(p, w, \alpha) \vdash (B, w, \alpha)$$

5. If B is a return block, then if ℓ appears on M_P's input and the label ℓ and block B' appear on the top of its stack, then M_P can read ℓ from the input, pop ℓ and B' off its stack, and go to state B':

   $$(B, \ell w, \ell B' \alpha) \vdash (B', w, \alpha)$$

6. Finally, M_P accepts by empty stack: for each state q, there is the move

   $$(q, \varepsilon, \$) \vdash (q, \varepsilon, \varepsilon).$$

We refer to the label appearing on the top of M_P's stack as the *current return label*, since this is the label of the program point to which control returns from a return block.

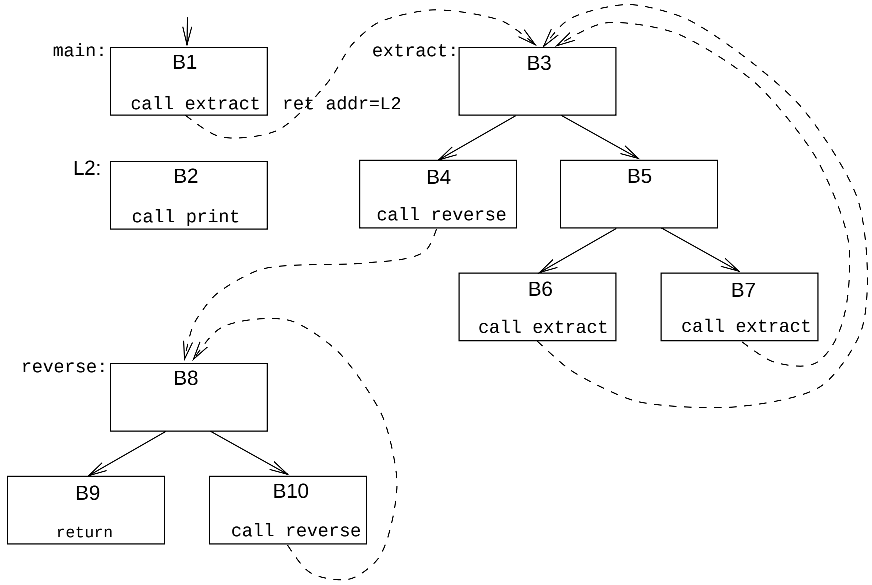

Figure 1: A (Partial) Flow Graph for the **main/extract/reverse** Program

4 Control Flow Grammars

Given a set of control flow graphs for the procedures in a program, we can construct a context free grammar that describes its control flow behavior. We call such a grammar a *control flow grammar*.

Definition 4.1 A control flow grammar for a program P is a context free grammar $G_P = (V, T, P, S)$ where the set of terminals T is the set Labels_P of return labels of P; the set of variables V is given by $V = \mathsf{Blocks}_P \cup \mathsf{Procs}_P$; the start symbol of the grammar is the entry procedure p of the program; and the productions of the grammar are given by the following:

1. if B is a basic block that is a predecessor of a basic block B', then there is a production $B \to B'$;
2. if B is a call block with return label ℓ, where the basic block labelled ℓ is B', and the called procedure is p, there is a production $B \to p\ \ell\ B'$;
3. if B is a tail call block and the called procedure is p, then there is a production $B \to p$;
4. if p is a procedure defined in P, and the control flow graph of p has entry node B, then there is a production $p \to B$.
5. If B is a return block, then there is a production $B \to \varepsilon$.

■

Example 4.1 Consider the **main/extract/reverse** program from Section 1. A (partial) flow graph for this program is shown in Figure 1, with ordinary control transfers shown using solid lines and calls to procedures using dashed lines. Because control transfers at procedure returns have not yet been determined, the predecessors of basic block B2 and the successors of block B9 are not yet known. The control flow grammar for this program has as its terminals the set of labels {L2}, nonterminals {**main**, **extract**, **reverse**, B1, ..., B10}, and the following productions:

`main`	$\rightarrow$	B1	B5	$\rightarrow$	B7
B1	$\rightarrow$	`extract` L2 B2	B6	$\rightarrow$	`extract`
B2	$\rightarrow$	`print`	B7	$\rightarrow$	`extract`
`extract`	$\rightarrow$	B3	`reverse`	$\rightarrow$	B8
B3	$\rightarrow$	B4	B8	$\rightarrow$	B9
B3	$\rightarrow$	B5	B8	$\rightarrow$	B10
B4	$\rightarrow$	`reverse`	B9	$\rightarrow$	ε
B5	$\rightarrow$	B6	B10	$\rightarrow$	`reverse`

The start symbol of the grammar is `main` □

The productions of the control flow grammar G_P closely resemble the moves of the control flow automaton M_P, and it comes as no surprise that they behave very similarly. Let $\Rightarrow_{lm}$ denote the leftmost derivation relation in G_P. The following theorem, whose proof closely resembles the standard proof of the equivalence between pushdown automata and context-free languages [6], expresses the intuition that the control flow grammar of a program mirrors the behavior of its control flow automaton:

Theorem 4.1 *Given a program P with entry point S, control flow grammar G_P and control flow automaton M_P,*

$$S \Rightarrow^{\star}_{lm} xA\beta \quad \textit{if and only if} \quad (S, xw, \$) \vdash^{\star} (A, w, \beta)$$

where $x, w \in \mathsf{Labels}_P^*$ and $A \in \mathsf{Blocks}_P \cup \mathsf{Procs}_P$.

5 Zeroth-Order Control Flow Analysis

Zeroth-order control flow analysis, also referred to as 0-CFA, involves determining, for each procedure p in a program, the set of labels $RetLbl(p)$ to which control can return after some call to p. Consider a call block B in a program P. If B is not a tail-call block, it pushes its return address onto the top of the stack before transferring control to the called procedure. On the other hand, if B is a tail call block, it leaves the control stack untouched and transfers control directly to the callee. Eventually, when executing a return, control branches to the label appearing on the top of the stack. Thus, in either case, the set of labels to which a procedure p can return is the set of current return labels when control enters p, in some configuration reachable from the initial configuration of M_P.

$$RetLbl(p) = \{\ell \mid (q_0, w, \$) \vdash^{\star} (p, w', \ell\beta)\}$$

It is a direct consequence of Theorem 4.1 that this set is precisely the FOLLOW set of p in the control flow grammar of the program (see [1] for the definition of FOLLOW sets):

Theorem 5.1 *For any procedure p in a program P, $RetLbl(p) = \mathrm{FOLLOW}(p)$.*

Proof Suppose the program P has entry point S. From the definition of $RetLbl(p)$, $\ell \in RetLbl(p)$ if and only if there is a call block A that calls p, such that $(S, xw, \$) \vdash^{\star} (A, w, \alpha) \vdash (p, w, \ell B\alpha)$. From Theorem 4.1, this is true if and only if $S \Rightarrow^{\star} xA\alpha \Rightarrow xp\ell B\alpha$, i.e., $S \Rightarrow^{\star} xp\ell\alpha'$. But this is true if only if $\ell \in \mathrm{FOLLOW}(p)$. It follows that $RetLbl(p) = \mathrm{FOLLOW}(p)$. □

Example 5.1 FOLLOW sets for some of the variables in the grammar of Example 4.1 are:

X	FOLLOW(X)
`main`	\$
`extract`	L2
`reverse`	L2

Here, a '\$' refers to the "external caller," e.g., the user. It is immediately apparent from this that control returns from `reverse` to the basic block in `main` that calls `print`. □

Input: A program P.

Output: The interprocedural 0-CFA control flow graph for P.

Method:

1. Construct the control flow grammar G_P for P.
2. Compute FOLLOW sets for the nonterminals of G_P.
3. Construct a partial control flow graph for P, without accounting for control transfers due to procedure returns. This is done by adding edges corresponding to intra-procedural control transfers, as in [1], together with an edge from each call block and tail-call block to the entry node of the corresponding called function.
4. For each $\ell \in \mathsf{Labels}_P$ and each nonterminal X of G_P do:

   if $\ell \in$ FOLLOW(X) and the basic block corresponding to X contains a `return` instruction then add an edge from X to B_ℓ, where B_ℓ is the basic block labelled by ℓ;

Figure 2: An algorithm for constructing the interprocedural 0-CFA control flow graph of a program

There is one remaining subtlety in constructing the interprocedural control flow graph of a program once the set of return labels for each function have been computed. If we consider the FOLLOW sets the grammar of Example 4.1, we find that L2 occurs in FOLLOW(`extract`), and it is correct to infer from this that control is transferred to the basic block B2, labelled by L2, after completion of the call to `extract`. However, we cannot conclude that each block that has L2 in its FOLLOW set has the block B2 as a successor: while L2 occurs in the FOLLOW sets of B3, B4, B5, B6, B7, B8, B9 and B10, it is not difficult to see that only B9—which actually contains a `return` instruction—should have B2 as a successor. The algorithm for constructing the control flow graph of a program, taking this into account, is given in Figure 2.

5.1 Applications of 0-CFA

An example application of 0-CFA is interprocedural unboxing optimization in languages that are either dynamically typed, or that support polymorphic typing. In implementations of such languages, the compiler cannot always predict the exact type of a variable at a program point, and as a result it becomes necessary to ensure that values of different types "look the same," which is achieved by "boxing." Unfortunately, manipulating boxed values is expensive.

The issue of maintaining untagged values has received considerable attention in recent years in the context of strongly typed polymorphic languages [5, 10, 13]. Using explicit "representation types," this work relies on the type system to propagate data representation information through the program. While theoretically elegant, the type system cannot be aware of low-level pragmatic concerns such as the costs of various representation conversion operations and the execution frequencies of different code fragments. As a result, it is difficult to guarantee that the "optimized" program is, in fact, more efficient than the unoptimized version. Also, the idea does not extend readily to dynamically typed languages.

Peterson [11] takes a procedure's control flow graph, and determines the optimal placement of representation conversion operations, based on basic block execution frequencies and conversion operation costs. As given, this is an intraprocedural optimization. For many programs, unboxing across procedure calls yields significant performance improvements. As an example, we tested a program that computes $\int_0^1 e^x dx$ using trapezoidal numerical integration with adaptive quadrature. For this program, intra-procedural unboxing optimization yields a performance improvement of about 30.3% (with a tail call from a function to itself being recognized and implemented as a loop). With inter-procedural unboxing, however, performance improves by about 52.9%.[1] To apply Peterson's algorithm interprocedurally, we need to construct the control flow graph

[1] We did not implement Peterson's algorithm, because the control flow analysis described here had not been developed when

for the entire program. The presence of tail call optimization makes computing the set of successors of a return block difficult with just the call graph of the program. Fortunately, 0-CFA provides precisely what is needed to determine the successors of return blocks, and thereby to construct the control flow graph for a program.

Another application of 0-CFA is interprocedural basic block fusion. The basic idea of this optimization is straightforward: if we have two basic blocks B_0 and B_1 where B_0 is the only predecessor of B_1 and B_1 is the only successor of B_0, then they can be combined into a single basic block; in the interprocedural case, the blocks being fused may belong to different procedures. The benefits of this optimization include a reduction in the number of jump instructions executed, with a concomitant decrease in pipeline "bubbles," as well as potentially improved opportunities for better instruction scheduling in the enlarged basic block resulting from the optimization. Our experience with filter fusion [14] indicates that this optimization can be of fundamental importance for performance in applications involving automatically generated source code.

Consider the `main/extract/reverse` program from Section 1. A partial flow graph for this program is given in Figure 1. It is not difficult to see that the 0-CFA algorithm of Figure 2 would determine that basic block B2 is the only successor of block B9, and B9 is the only predecessor of B2, thereby allowing these two blocks to be fused. Note that a naive analysis that handles tail calls as if they were calls that returned to an empty basic block immediately following the call site would infer that basic block B2 had three predecessors, blocks B4, B6 and B7, thereby preventing the application of the optimization in this case.

6 First-Order Control Flow Analysis

While 0-CFA tells us the possible return addresses for each basic block and procedure, it leaves out all information about the "context" in which a call occurs (i.e., who called whom to get to this call site). This may render 0-CFA inadequate in some situations. Information about where control "came from" could provide more precise liveness or aliasing information at a particular program point, allowing a compiler to generate better code.

At any point during a program P's execution, the return addresses on its control stack (which correspond to the contents of the stack in some execution of the control flow automaton M_P) give us a complete history of the interprocedural control flow behavior of the program upto that point. Since, the set of all possible (finite) sequences of labels is infinite, we seek finitely computable approximations to this information. An obvious possibility is to keep track of the top k labels on the stack of M_P, for some fixed $k \geq 0$. 0-CFA, where we keep track of no context information at all, corresponds to choosing $k = 0$. A control flow analysis that keeps track of the top k return addresses on the stack of M_P is called a *k-th.-order control flow analysis*, or k-CFA (this corresponds to the "call-strings approach" of Sharir and Pnueli [16]). In this section, we focus our attention on first-order control flow analysis, or 1-CFA.

In the previous section, we showed that the FOLLOW sets of the control flow grammar give 0-CFA information. How might we incorporate additional context information into such analyses? In parsing theory, FOLLOW sets are used to construct SLR(1) parsers, which are based on LR(0) items. Because SLR(1) parsers do not maintain much context information, they are unable to handle many simple grammars. Introducing additional context information into the items using lookahead tokens fixes this problem: this leads to the use of LR(1) items.

This analogy carries over to control flow analysis. LR(1) items for the control flow grammar G_P are closely related to the information manipulated during 1-CFA. Basically, an LR(1) item $[A \to \alpha \circ \beta,\ a]$ conveys the information that control can reach A with current return label a. In an item $[A \to \alpha \circ \beta,\ a]$ we will often focus on the nonterminal A on the left hand side of the production and the lookahead token a, but not in the details of the structure of $\alpha \circ \beta$: in such cases, to reduce visual clutter we will write the item as $[A \to \cdots,\ a]$. In the context of this discussion we are not concerned with whether or not the control flow grammar G_P is LR(1)-parsable.

We know, from parsing theory, that given a control flow grammar G_P with variables V and terminals T, there is a nondeterministic finite automaton (NFA) $(Q, \Sigma, \delta, q_0, Q)$ that recognizes viable prefixes of G [6].

we implemented the unboxing optimization. Instead, our compiler uses a heuristic that produces good, but not necessarily optimal, representation conversion placements. These performance improvements can therefore be seen as lower bounds on what can be attained using an optimal algorithm.

This NFA, which we will refer to as the *viable prefix NFA*, is defined as follows: its set of states Q consists of the set of LR(1) items for G_P, together with a state q_0 that is not an item; its alphabet $\Sigma = V \cup T$; the initial state is q_0; every state is a final state; and the transition function δ is defined as follows:

(*i*) Given a program P with entry point p, $\delta(q_0, \varepsilon) = \{[S \to \circ\, p,\ \$]\}$.
(*ii*) $\delta([A \to \alpha \circ B\beta,\ a], \varepsilon) = \{[B \to \circ\, \gamma,\ b] \mid B \to \gamma \text{ is a production}, b \in \mathrm{FIRST}(\beta a)\}$.
(*iii*) $\delta([A \to \alpha \circ B\beta,\ a], B) = \{[A \to \alpha B \circ \beta,\ a]\} \qquad (B \neq \varepsilon)$.

An item I is said to be *reachable* if $q_0 \rightsquigarrow^\star I$, i.e., if there is a path from the the initial state q_0 of the viable prefix NFA to I. The following result makes explicit the correspondence between LR(1) items in G_P and return addresses on top of the stack of M_P:

Theorem 6.1 *Given a program with entry point p,* $(p, x, \$) \vdash^\star (A, y, a\alpha)$ *if and only if there is a reachable item* $[A \to \cdots,\ a]$.

The set of current return labels, i.e., labels at the top of M_P's stack, when control enters a basic block or procedure is now easy to determine:

Corollary 6.2 *Let A be any basic block or procedure in a program P. The set of current return labels when control enters A is given by* $\{\ell \mid$ *there is a reachable LR(1) item* $[A \to \cdots,\ \ell]\}$.

An LR(1) item $[A \to \cdots,\ a]$ tells us about the return labels that can appear on top of the control stack, i.e., about addresses that control can go to. Fortunately, it turns out that we can use the reachability relation $\rightsquigarrow$ in the viable prefix NFA to trace the origins of a call. Consider a program P with a call block A that calls a procedure p: in the control flow grammar G_P, this corresponds to a production

$$A \to p\ \ell\ C$$

where ℓ is the return label and C is the block with label ℓ. This production gives rise to LR(1) items of the form $[A \to\ \circ\, p\ \ell\ C,\ b]$. Let B_p be the entry node of the flow graph for p, then G_P contains the production $p \to B_p$, so in the viable prefix NFA there is a ε-transition from each of these items to the item $[p \to\ \circ\, B_p,\ \ell]$. Suppose the block B_p has successors $C_1, \ldots, C_k$, then G_P has productions $B_p \to C_1, \cdots B_p \to C_k$, and the viable prefix NFA will therefore have ε-transitions from the item $[p \to\ \circ\, B_p,\ \ell]$ to each of the items $[B_p \to\ \circ\, C_1,\ \ell], \ldots, [B_p \to\ \circ\, C_k,\ \ell]$. Suppose one of these blocks, say C_j, makes a tail call to a procedure q, whose flow graph has entry node B_q, then G_P contains the productions $C_j \to q$ and $q \to B_q$, and this gives rise to ε-transitions from $[B_p \to\ \circ\, C_j,\ \ell]$ to $[C_j \to\ \circ\, q,\ \ell]$ and thence to $[q \to\ \circ\, B_q,\ \ell]$. We can follow ε-transitions in this way to trace a sequence of control transfers that does not involve any procedure returns. Conversely, we can follow ε-transitions backwards from a call to work out where it could have come from.

Intuitively, we want to be able to characterize a collection of successive basic blocks and procedures control can go through—i.e., a sequence of states of the control flow automaton—without any procedure returns, except perhaps at the very end. Since the set of sequences of blocks is infinite, we need a finite approximation: as before, one simple way to do this is to consider sets of blocks (there are only finitely many), together with the current return label when control reaches each block. These ideas can be made more precise using the notion of a *forward chain*:

Definition 6.1 A forward chain in a program P is a set $\{(B_0, \ell_0) \ldots, (B_n, \ell_n)\}$ where each B_i is either a procedure in P or a basic block in P, $\ell_i \in \mathsf{Labels}_P$ for $0 \leq i \leq n$, and where for each i, $0 \leq i < n$, the following hold: (*i*) B_i is not a return block; and (*ii*) in the control flow automaton M_P, $(B_i, x, \ell_i\alpha) \vdash (B_{i+1}, y, \ell_{i+1}\beta)$ for some x, y, α, β. ■

Reasoning as above, it is easy to show the following result:

Theorem 6.3 $\{(B_0, \ell_0), \ldots, (B_n, \ell_n)\}$ *is a forward chain in a program P if and only if there is a sequence of ε-transitions in the viable prefix NFA for* G_P *of the form*

$$\begin{aligned}
[B_0 \to \alpha \circ B_1\beta_0,\ \ell_0] &\rightsquigarrow [B_1 \to\ \circ\, B_2\beta_1,\ \ell_1] \\
&\rightsquigarrow \cdots \\
&\rightsquigarrow [B_{n-1} \to\ \circ\, B_n\beta_{n-1},\ \ell_{n-1}] \\
&\rightsquigarrow [B_n \to\ \circ\, \beta_n,\ \ell_n]
\end{aligned}$$

where $\ell_{i+1} \in \mathrm{FIRST}(\beta_i \ell_i)$*, for some* $\beta_0, \ldots, \beta_n$*.*

Now consider the process of applying the subset construction to the viable prefix NFA to construct an equivalent DFA. Each state of the DFA consists of a set of NFA states—that is, a set of LR(1) items—obtained by starting with a set of NFA states and then adding all the states reachable using only ε-transitions. The DFA construction is useful because the set of NFA states comprising each state of the DFA corresponds to the largest set of NFA states reachable from some initial set using only ε-transitions, i.e., to a set of maximal-length forward chains. In other words, if a forward chain occurs in a state of the ciable prefix DFA, it is entirely contained in that state: it can can never spill over into another state, thereby simplifying the search for forward chains. This is expressed by the following result:

Corollary 6.4 $\{(B_0, \ell_0), \ldots, (B_n, \ell_n)\}$ *is a forward chain in a program if and only if there is a state in the viable prefix DFA for* G_P *containing items* $[B_0 \to \alpha \circ B_1 \beta_0,\ \ell_0]$, $[B_1 \to\ \circ B_2 \beta_1,\ \ell_1]$, $\cdots$, $[B_{n-1} \to\ \circ B_n \beta_{n-1},\ \ell_{n-1}]$, $[B_n \to \beta_n,\ \ell_n]$ *where* $\ell_{i+1} \in \mathrm{FIRST}(\beta_i \ell_i)$*, for some* $\beta_0, \ldots, \beta_n$*.*

Intuitively, control can "come from" a call A to a point B if there is a forward chain from A to B that contains no intervening calls—i.e., A is the most recent call preceding B. The following result is now immediate:

Corollary 6.5 *Let* A *be a call block or a tail call block in a program* P*, and* B *a basic block or a procedure in* P*. Then, control can come from* A *to* B *if and only if there is a state in the viable prefix DFA of* G_P *containing items* $[B_0 \to \alpha \circ B_1 \beta_0,\ \ell_0]$, $[B_1 \to\ \circ B_2 \beta_1,\ \ell_1]$, $\cdots$, $[B_{n-1} \to\ \circ B_n \beta_{n-1},\ \ell_{n-1}]$, $[B_n \to \beta_n,\ \ell_n]$ *where* $\ell_{i+1} \in \mathrm{FIRST}(\beta_i \ell_i)$*, for some* $\beta_0, \ldots, \beta_n$*, such that* $B_0 \equiv A$*,* $B_n \equiv B$*, and* B_i *is not a call block or tail call block for* $0 < i < n$*.*

We conjecture that the analogy between control flow analysis and LR items continues to hold when more context information is maintained. In particular, we conjecture that just as first-order control flow analysis (1-CFA) corresponds to LR(1) items, k-th.-order control flow analysis (k-CFA) corresponds to LR(k) items.

6.1 Applications of 1-CFA

An example application of 1-CFA is in context-sensitive interprocedural dataflow analysis. Much of the recent work on interprocedural dataflow analysis has focused on languages such as C and Fortran, whose implementations usually do not support tail call optimization. These analyses determine, for each call, the behavior of the called procedure, then propagate this information to the program point to which that call returns. For the languages considered, the determination of the return points for the calls is straightforward. Because the point to which a call returns is not obvious in the presence of tail call optimization, it is not obvious how to apply these analyses to systems with tail call optimization. While 0-CFA can be used to determine the set of successors for each return block, this does not maintain enough context information to determine where control came from. As a result, the analysis can infer spurious pointer aliases by propagating information from one call site back to a different call site. Using *context-sensitive interprocedural analyses* avoids this by maintaining information about where a call came from [2, 8, 22], which is precisely the information provided by 1-CFA.

As a specific example of the utility of context-sensitive flow information, our experiments with dead code elimination based on interprocedural liveness analysis, in the context of the alto link-time optimizer [3] applied to a number of SPEC benchmarks, indicate that compared to the number of register loads and stores that can be deleted based on context-insensitive liveness information, an additional 5%–8% can be deleted using context-sensitive liveness information.

Whether or not a context-sensitive version of an interprocedural analysis is useful depends, to a great extent, on the analysis and the application under consideration. Our experiments with interprocedural liveness analysis indicate that there are situations when such analyses can lead to a noticeable improvement in the code generated. On the other hand, in comparing context-sensitive and context-insensitive alias analyses due to indirect memory references through pointers, Ruf observes [15] that "...the context-sensitive analysis does compute more precise alias relationships at some program points. However, when we restrict our attention to the locations accessed by or modified by indirect memory references, no additional precision is measured." However, *if* a context-sensitive dataflow analysis is deemed necessary for a language implementation with tail call optimization, the control flow analysis described here can be used to provide the necessary support.

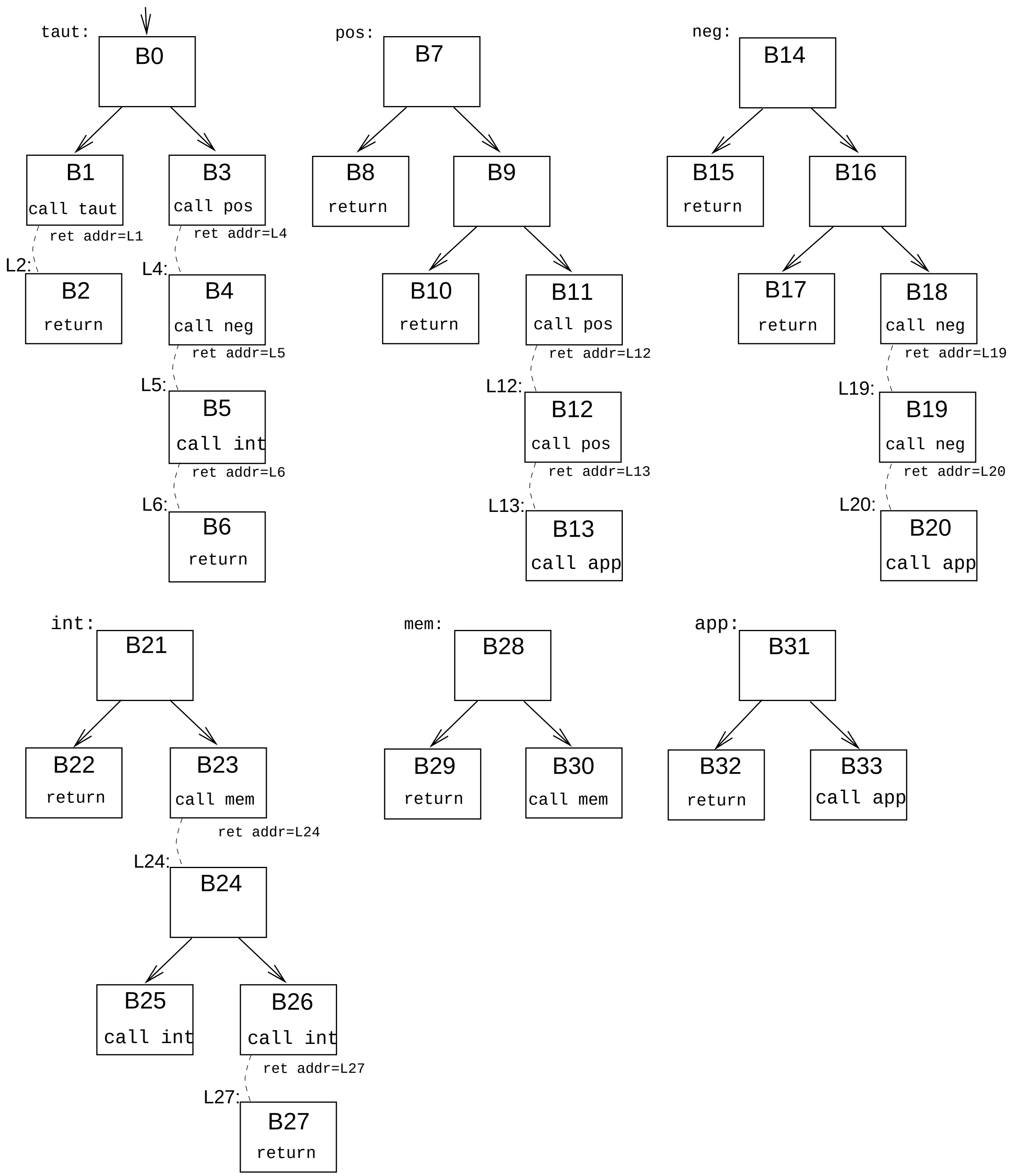

Figure 3: A (Partial) Flow Graph for the Tautology Checker Program

taut	$\rightarrow$	B0	B12	$\rightarrow$	pos L13 B13	B16	$\rightarrow$	B18
B0	$\rightarrow$	B1	B13	$\rightarrow$	app	B25	$\rightarrow$	int
B0	$\rightarrow$	B3	neg	$\rightarrow$	B14	B26	$\rightarrow$	int L27 B27
B1	$\rightarrow$	taut L2 B2	B14	$\rightarrow$	B15	B27	$\rightarrow$	ε
B2	$\rightarrow$	ε	B14	$\rightarrow$	B16	mem	$\rightarrow$	B28
B3	$\rightarrow$	pos, L4 B4	B15	$\rightarrow$	ε	B28	$\rightarrow$	B29
B4	$\rightarrow$	neg L5 B5	B16	$\rightarrow$	B17	B28	$\rightarrow$	B30
B5	$\rightarrow$	int L6 B6	B17	$\rightarrow$	ε	B29	$\rightarrow$	ε
B6	$\rightarrow$	ε	B18	$\rightarrow$	neg L19 B19	B30	$\rightarrow$	mem
pos	$\rightarrow$	B7	B19	$\rightarrow$	neg L20 B20	app	$\rightarrow$	B31
B7	$\rightarrow$	B8	B20	$\rightarrow$	app	B31	$\rightarrow$	B32
B7	$\rightarrow$	B9	int	$\rightarrow$	B21	B31	$\rightarrow$	B33
B8	$\rightarrow$	ε	B21	$\rightarrow$	B22	B32	$\rightarrow$	ε
B9	$\rightarrow$	B10	B21	$\rightarrow$	B23	B33	$\rightarrow$	append
B9	$\rightarrow$	B11	B22	$\rightarrow$	ε	B23	$\rightarrow$	mem L24 B24
B10	$\rightarrow$	ε	B24	$\rightarrow$	B25	B24	$\rightarrow$	B26
B11	$\rightarrow$	pos L12 B12						

Figure 4: Productions for the control flow grammar of the program in Section 7

7 A Larger Example

Consider the following program, adapted from Section 4.17 of [12], to determine whether a propositional formula in conjunctive normal form is a tautology:

```
fun taut(Conj(p,q)) = taut(p) andalso taut(q)
  | taut(p) = ( [] <> int(pos(p), neg(p)) );
fun pos(Atom(a)) = [a]
  | pos(Neg(Atom(a))) = []
  | pos(Disj(p,q)) = app(pos(p), pos(q));
fun neg(Atom(a)) = []
  | neg(Neg(Atom(a))) = [a]
  | neg(Disj(p,q)) = app(neg(p), neg(q));
fun int([], ys) = []
  | int(x::xs, ys) = if mem(x, ys) then x :: int(xs, ys) else int(xs, ys);
fun mem(x, []) = false
  | mem(x, y::ys) = (x=y) orelse mem(x, ys);
fun app([], ys) = ys
  | app(x::xs, ys) = x::app(xs, ys);
```

The partial flow graph for this program is shown in Figure 3. To reduce clutter, we have not explicitly shown control transfers due to procedure calls; moreover, to aid the reader in understanding the control flow behavior of this program, each non-tail call is connected to the basic block corresponding to its return address with a dashed arc. The control flow grammar $G = (V, T, P, S)$ for this program is given by the following: $V = \{$taut, pos, neg, int, mem, app, B0, …, B33$\}$; $T = \{$L2, L4, L5, L6, L12, L13, L19, L20, L24, L27$\}$; $S =$ taut; and the set of productions P as shown in Figure 4. The set of possible return addresses for each function, as obtained using 0-CFA, is as follows:

taut :	L2, \$
pos :	L4, L12, L13
neg :	L5, L19, L20
int :	L6, L27
mem :	L24
app :	L4, L12, L13, L5, L19, L20

Due to space constraints, we do not reproduce all the sets of LR(1) items for this grammar. The difference between 0-CFA and 1-CFA can be illustrated by examining the behavior of the function app. On examining the viable prefix DFA, we find four states that are relevant to this function. One of these states consists of the following two groups of LR(1) items:

$[B12 \to \mathsf{pos}\ \mathsf{L13} \circ B13,\ \mathsf{L4}]$	$[B12 \to \mathsf{pos}\ \mathsf{L13} \circ B13,\ \mathsf{L12}]$
$[B13 \to\ \circ\, \mathsf{app},\ \mathsf{L4}]$	$[B13 \to\ \circ\, \mathsf{app},\ \mathsf{L12}]$
$[\mathsf{app} \to\ \circ\, B31,\ \mathsf{L4}]$	$[\mathsf{app} \to\ \circ\, B31,\ \mathsf{L12}]$
$[B31 \to\ \circ\, B32,\ \mathsf{L4}]$	$[B31 \to\ \circ\, B32,\ \mathsf{L12}]$
$[B31 \to\ \circ\, B33,\ \mathsf{L4}]$	$[B31 \to\ \circ\, B33,\ \mathsf{L12}]$
$[B32 \to\ \circ\, ,\ \mathsf{L4}]$	$[B32 \to\ \circ\, ,\ \mathsf{L12}]$
$[B33 \to\ \circ\, \mathsf{app},\ \mathsf{L4}]$	$[B33 \to\ \circ\, \mathsf{app},\ \mathsf{L12}]$

From the forward chains in the first group, we can determine that app can be called from basic block B13 of the function pos, with return label L4 (note that this refers to a block that—because of the control flow effects of tail-call optimization—does not belong to the calling function), and this can then recursively call itself with the same return label. In this case, the return label indicates that the calling function pos was itself called from taut. The second group of LR(1) items shows a similar call sequence to app from basic block B13, except that in this case the calling function pos is being called recursively from basic block B11 in the body of pos. The remaining three states relevant to the function app provide similar information: one of these contains two groups of items, the first of which is identical to the first group above, and the second of which is similar to the second group above except that it refers to a recursive call to pos from basic block B12; the remaining two states provide similar information for calls to app from the function neg.

8 Trading Precision for Efficiency

One of the biggest advantages we see for a grammatical formulation of control flow analysis is that grammars and parsing have been studied extensively and are generally well understood. Because of this, a wide variety of techniques and tools originally devised for syntax analysis are applicable to control flow analysis.

As an example of this, consider the fact that the efficiency of compile time analyses can be improved by reducing the amount of information maintained and manipulated, i.e., by decreasing precision. In the case of control flow analysis, determining where control came from involves examining the states of the viable prefix DFA of the control flow grammar, constructed using LR(1) items. It is well-known that the number of states in such a DFA can become very large, but that by judiciously merging certain states (those with a common "kernel", see [1]) the number of states can be reduced considerably without significantly sacrificing the information contained in the DFA. Parsers that are constructed in this way are known as LALR(1) parsers, which can be built efficiently (without initially building the LR(1) DFA).

It does not come as a surprise that the same idea can be applied to 1-CFA as well. The resulting analysis is more precise than 0-CFA and potentially somewhat less precise than 1-CFA: with tongue firmly in cheek, we call such an analysis $\frac{1}{2}$-CFA. It is usually considerably more efficient than 1-CFA. As an example, for the tautology checker program of Section 7, the viable prefix DFA constructed from LR(1) items contains 97 states, while that constructed from LALR(1) items contains 55 states; if we consider the entire tautology checker from [12], which works for arbitrary propositional formulae, the LR(1) viable prefix DFA has 304 states while the LALR(1) DFA has 112 states. If we focus on calls to the function app, as in Section 7, we find that with LALR(1) items it suffices to examine a single state of the DFA, in contrast to four states for the LR(1) case. Moreover, there is no loss of information regarding the calling contexts in this case.

9 Conclusions

Knowledge of low-level control flow is essential for many compiler optimizations. In systems with tail call optimization, the determination of interprocedural control flow is complicated by the fact that because of tail call optimization, control flow at procedure returns is not readily evident from the call graph of the program. In this paper, we show how interprocedural control flow analysis of first-order programs can be carried out using well-known concepts from parsing theory. In particular, we show that 0-CFA corresponds to the notion of FOLLOW sets in context free grammars, and 1-CFA corresponds to the analysis of LR(1) items. The control flow information so obtained can be used to improve the precision of interprocedural dataflow analyses as well as to extend certain low-level code optimizations across procedure boundaries.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] J.-D. Choi, M. Burke, and P. Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects", *Proc. 20th. ACM Symposium on Principles of Programming Languages*, Jan. 1993, pp. 232–245.

[3] K. De Bosschere and S. K. Debray, "`alto` : A Link-Time Optimizer for the DEC Alpha", Technical Report 96-15, Dept. of Computer Science, The University of Arizona, Tucson, June 1996.

[4] N. Heintze, "Control Flow Analysis and Type Systems", Technical Report CMU-CS-94-227, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Dec. 1994.

[5] F. Henglein and J. Jørgensen, "Formally Optimal Boxing", *Proc. 21st. ACM Symp. on Principles of Programming Languages*, Portland, OR, Jan. 1994, pp. 213–226.

[6] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 1979.

[7] S. Jagannathan and S. Weeks, "A Unified Treatment of Flow Analysis in Higher-Order Languages", *Proc. 22nd. ACM Symp. on Principles of Programming Languages*, San Francisco, Jan. 1995, pp. 393–407.

[8] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing", *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992, pp. 235–248.

[9] T. Lindgren, "Control Flow Analysis of Prolog", *Proc. 1995 International Symposium on Logic Programming*, Dec. 1995, pp. 432–446. MIT Press.

[10] X. Leroy, "Unboxed objects and polymorphic typing", *Proc. 19th. ACM Symp. on Principles of Programming Languages*, Albuquerque, NM, Jan. 1992, pp. 177–188.

[11] J. C. Peterson, "Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time", *Proc. Functional Programming Languages and Computer Architecture*, London, Sept. 1989, pp. 89–99.

[12] L. C. Paulson, *ML for the Working Programmer*, Cambridge University Press, 1991.

[13] S. Peyton Jones and J. Launchbury, "Unboxed values as first class citizens in a non-strict functional language", *Proc. Functional Programming Languages and Computer Architecture 1991*, pp. 636–666.

[14] T. A. Proebsting and S. A. Watterson, "Filter Fusion", *Proc. 23rd. ACM Symposium on Principles of Programming Languages*, Jan. 1996, pp. 119–129.

[15] E. Ruf, "Context-Insensitive Alias Analysis Reconsidered", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 13–22.

[16] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Dataflow Analysis", in *Program Flow Analysis: Theory and Applications*, eds. S. S. Muchnick and N. D. Jones, Prentice-Hall, 1981, pp. 189–233.

[17] O. Shivers, "Control Flow Analysis in Scheme", *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 164–174.

[18] O. Shivers, *Control Flow Analysis of Higher-Order Languages*, PhD. Dissertation, Carnegie Mellon University, May 1991. Also available as Technical Report CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991.

[19] Y. Tang and P. Jouvelot, "Control-Flow Effects for Escape Analysis", *Proc. WSA 92*, Bordeaux, France, 1992.

[20] Y. Tang and P. Jouvelot, "Separate Abstract Interpretation for Control Flow Analysis", *Proc. TACS-94*, 1994.

[21] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee, "TIL: A Type-Directed Optimizing Compiler for ML", *Proc. SIGPLAN '96 Conference on Programming Language Design and Implementation*, June 1996, pp. 181–192.

[22] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 1–12.

A Proofs of Main Theorems

Theorem 4.1 *Given a program P with entry point S, control flow grammar G_P and control flow automaton M_P,*

$$S \Rightarrow^{\star}_{lm} xA\beta \quad \text{if and only if} \quad (S, xw, \$) \vdash^{\star} (A, w, \beta)$$

where $x, w \in \mathsf{Labels}_P^*$ and $A \in \mathsf{Blocks}_P \cup \mathsf{Procs}_P$.

Proof: We prove a slightly stronger result, namely, that for all $i \geq 0$, $S \Rightarrow^{i}_{lm} xA\beta$ if and only if $(S, xw, \$) \vdash^{i} (A, w, \beta)$. The proof is by induction on i. The base case, with $i = 0$, is trivial, with $S = A$, $x = \varepsilon$ and $\beta = \varepsilon$. For the inductive case, assume that the theorem holds for derivations of length upto i, and consider a derivation of length $i + 1$. This must have the form

$$S \Rightarrow^{i}_{lm} yB\alpha \Rightarrow xA\beta$$

where $x, y \in \mathsf{Labels}_P^*$. Since $S \Rightarrow^{i}_{lm} yB\alpha$, from the induction hypothesis we have $(S, yw, \$) \vdash^{i} (B, w, \alpha)$. Consider the last step in the leftmost derivation in G_P shown above. Depending on the production used in this step, we have the following cases:

Case 1 : This covers the cases where either B is not a call block, and has basic block A as a successor; or B is a tail call block that calls procedure A; or B is a procedure whose flow graph entry node is A. In each case, the production used has the form $B \rightarrow A$ where A is a nonterminal, and the definition of M_P indicates that the only move M_P can make is to change its state from B to A, leaving its input and stack untouched. Thus, $y = x$ and $\alpha = \beta$, and we have $(S, xw, \$) \vdash^{i} (B, w, \alpha) \vdash (A, w, \alpha)$. It follows from this that $S \Rightarrow^{i+1}_{lm} xA\beta$ iff $(S, xw, \$) \vdash^{i+1} (A, w, \beta)$.

Case 2 : B is a call block. In this case, A must be the called procedure. Let ℓ be the return label and B' the basic block with label ℓ. The production used in the derivation of G_P must be

$$B \rightarrow A \ \ell \ B'$$

which yields the derivation $S \Rightarrow^{i}_{lm} yB\alpha \Rightarrow xA\ell B'\alpha$ i.e., $\beta = \ell B'\alpha$. The corresponding move of M_P must be

$$(B, w, \alpha) \vdash (A, w, \ell B'\alpha) = (A, w, \beta).$$

Thus, we have $S \Rightarrow^{i+1}_{lm} xA\beta$ iff $(S, xw, \$) \vdash^{i+1} (A, w, \beta)$.

Case 3 : B is a return block. The production used in the derivation of G_P is $B \rightarrow \varepsilon$, and the resulting derivation is

$$S \Rightarrow^{i}_{lm} yB\alpha \Rightarrow x\alpha.$$

It is straightforward to show, by induction on the length of derivations, that α must be of the form $\ell_1 B_1 \ell_2 B_2 \cdots \ell_n B_n$, where ℓ_1 is a terminal and B_1 is a nonterminal. Thus, $x = y\ell_1$, $A = B_1$, and $\beta = \ell_2 B_2 \cdots \ell_n B_n$. So $\alpha = \ell_1 A\beta$. The corresponding move of M_P is

$$(B, w, \alpha) = (B, w, \ell_1 A\beta) \vdash (A, w, \beta).$$

Thus, we have $S \Rightarrow^{i+1}_{lm} xA\beta$ iff $(S, xw, \$) \vdash^{i+1} (A, w, \beta)$.

These cases exhaust all the possibilities. The theorem follows. $\Box$

Theorem 6.1 *Given a program with entry point p, $(p, x, \$) \vdash^{\star} (A, y, a\alpha)$ if and only if there is a reachable item $[A \rightarrow \cdots,\ a]$.*

Proof: [$\Rightarrow$] We show, by induction on i, that if $(p, x, \$) \vdash^{i} (A, y, a\alpha)$ then $q_0 \leadsto^{\star} [A \rightarrow \cdots,\ a]$. The base case is for $i = 0$: in this case, $p \equiv A$, $x \equiv y$, $a \equiv \$$ and $\alpha \equiv \varepsilon$. There is a production $p \rightarrow B$ in the control flow grammar G_P, where B is the entry node of the flow graph of p. It follows that there is a transition from q_0 to $[p \rightarrow \ \circ B,\ \$]$ in the viable prefix NFA.

Assume that $(p, x, \$) \vdash^{j} (A, y, a\alpha)$ implies $q_0 \leadsto^{\star} [A \rightarrow \cdots,\ a]$ for all $j \leq i$, and consider a sequence of moves

$$(p, x, \$) \vdash^i (B, y, b\beta) \vdash (A, z, a\alpha).$$

Consider the last move, $(B, y, b\beta) \vdash (A, z, a\alpha)$. We have the following cases, depending on the nature of B:

Case 1 : B is a call block where the called procedure is A, the return label is a, and the block with label a is C. In this case, $y \equiv z$, and the last move becomes

$$(B, y, b\beta) \vdash (A, y, aCb\beta).$$

In other words, $(B, y, b\beta) \vdash (A, y, a\alpha)$, where $\alpha \equiv Cb\beta$. In the control flow grammar G_P, there is a production

$$B \to AaC.$$

Now we have, from the induction hypothesis, that $q_0 \leadsto^\star [B \to \cdots,\ b]$. This can happen if and only if there is some item I of the form $[X \to \zeta \circ B\eta,\ c]$ such that $q_0 \leadsto^\star I$. Since $B \to AaC$ is a production, this means that we can have the transition sequence

$$\begin{aligned} q_0 &\leadsto^\star [X \to \zeta \circ B\eta,\ c] \\ &\leadsto [B \to\ \circ AaC,\ d], \quad d \in \mathrm{FIRST}(\eta c) \\ &\leadsto [A \to \cdots,\ a] \end{aligned}$$

Thus, $(p, x, \$) \vdash^{i+1} (A, y, a\alpha)$ implies $q_0 \leadsto^\star [A \to \cdots,\ a]$.

Case 2 : B is a return block. The moves of M_P must be of the form

$$(p, x, \$) \vdash^i (B, \ell z, \ell A\alpha) \vdash (A, z, \alpha).$$

Now consider how ℓA came to be on M_P's stack: there must have been a call block C that called a procedure p with return label ℓ, where the basic block with label ℓ is A. This means that there is a sequence of moves

$$(p, x, \$) \vdash^j (C, u, a\gamma) \vdash (p, u, \ell Aa\gamma)$$

where $j < i$. Since C is not a tail-call block, our assumption of well-behavedness of programs implies that control returns to the return address ℓ (i.e., the block A) when the execution of the call has completed, and that the stack at that point is $a\gamma$. From the induction hypothesis,

$$\begin{aligned} q_0 &\leadsto^\star [C \to\ \circ p\ell A,\ a] \\ &\leadsto [C \to p \circ \ell A,\ a] \\ &\leadsto [C \to p\ell \circ A,\ a] \\ &\leadsto [A \to \cdots,\ a]. \end{aligned}$$

Thus, $(p, x, \$) \vdash^{i+1} (A, z, a)$ implies $q_0 \leadsto^\star [A \to \cdots,\ a]$.

Case 3 : B is not a call block, or B is a procedure and A is the entry node in the flow graph of B, or B is a tail call block and A the tail-called procedure. The corresponding production in G_P is $B \to A$. This case corresponds to a simple transfer of control that does not affect the stack. So the moves of M_P are

$$(p, x, \$) \vdash^i (B, z, a\alpha) \vdash (A, z, a\alpha).$$

As in Case 1, $q_0 \leadsto^\star [B \to \cdots,\ b]$ implies that there is an item I of the form $[X \to \zeta \circ B\eta,\ c]$ such that $q_0 \leadsto^\star I$. Since $B \to A$ is a production, it follows from the induction hypothesis that $q_0 \leadsto^\star I \leadsto [B \to\ \circ A,\ a] \leadsto [A \to \cdots,\ a]$.

This establishes that for all $i \geq 0$, $(p, x, \$) \vdash^i (A, z, a\alpha)$ implies $q_0 \leadsto^\star [A \to \cdots,\ a]$.

[$\Leftarrow$] We use the notation $A \leadsto^k B$ to denote that there is a path of length k from state A to state B of a viable prefix NFA. We show, by induction on i, that $q_0 \leadsto^i [A \to \cdots,\ a]$ implies $(p, x, \$) \vdash^\star (A, z, a\alpha)$. The base case, with $i = 1$, is straightforward: in this case $A \equiv p$ is the entry point of the program and $z \equiv \$$.[2]

[2] The reason the base case is for $i = 1$ is that the viable prefix NFA contains an initial state q_0 that does not correspond to any item: thus, it requires a single ε-transition to reach an item.

There is a transition from q_0 to the item $[A \to \circ B,\ \$]$ where B is the entry node of the flowgraph of A, and $(A, x, \$)$ is the initial configuration of M_P.

Assume that $q_0 \rightsquigarrow^j [A \to \cdots,\ a]$ implies $(p, x, \$) \vdash^\star (A, y, a\alpha)$ for all $j \leq i$, and consider a transition sequence

$$q_0 \rightsquigarrow^i I_0 : [B \to \cdots,\ b] \rightsquigarrow I_1 : [A \to \cdots,\ a].$$

Consider the last transition of the viable prefix NFA: either it is an ε-transition, or it is not. If it is an ε-transition, B cannot be a return block, since a return block X corresponds to an item of the form $[X \to \circ\,,\ a]$, and the corresponding state in the viable prefix NFA has no outgoing transitions. We therefore have the following cases, depending on the nature of B:

Case 1 : B is a call block. In this case, given the definition of G_P, the item I_0 must be of the form $[B \to \circ AaC,\ b]$, where a is the return label for this call and C is the basic block with label a. This yields the transition sequence

$$q_0 \rightsquigarrow^i I_0 : [B \to \circ AaC,\ b] \rightsquigarrow I_1 : [A \to \cdots,\ a].$$

The lookahead symbol of I_1 is obtained from $\text{FIRST}(aCb) = \{a\}$. From the induction hypothesis, $(p, x, \$) \vdash^\star (B, z, b\beta)$, and since B is a call block, the next move of M_P is $(B, z, b\beta) \vdash (A, z, aCb\beta)$. Thus, we have

$$(p, x, \$) \vdash^\star (B, z, b\beta) \vdash (A, z, a\alpha), \quad \text{where } \alpha = Cb\beta.$$

Case 2 : B is not a call block, i.e., it has no effect on the stack. The corresponding grammar productions are of the form $B \to A$, and we have

$$q_0 \rightsquigarrow^i [B \to \circ A,\ a] \rightsquigarrow [A \to \cdots,\ a].$$

From the induction hypothesis, $(p, x, \$) \vdash^\star (B, z, a\alpha)$, and from the definition of M_P, $(B, z, a\alpha) \vdash (A, z, a\alpha)$. Thus, M_P makes the moves

$$(p, x, \$) \vdash^\star (B, z, a\alpha) \vdash (A, z, a\alpha).$$

Thus, $q_0 \rightsquigarrow^{i+1} [A \to \cdots,\ a]$ implies $(p, x, \$) \vdash^\star (A, z, a\alpha)$.

On the other hand, if the last transition is not an ε-transition, the items I_0 and I_1 must have the form $I_0 = [A \to \gamma \circ X\psi,\ a]$ and $I_1 = [A \to \gamma X \circ \psi,\ a]$, for some $\gamma, \psi \in (V \cup T)^*$. In this case, the transitions of the viable prefix NFA must be of the form

$$q_0 \rightsquigarrow^i I_0 : [A \to \gamma \circ X\psi,\ a] \rightsquigarrow I_1 : [A \to \gamma X \circ \psi,\ a].$$

Then, since $q_0 \rightsquigarrow^i I_0 : [A \to \gamma \circ X\psi,\ a]$, the induction hypothesis implies that $(p, x, \$) \vdash^\star (A, z, a\alpha)$. This establishes that for all $i \geq 0$, $q_0 \rightsquigarrow^i [A \to \cdots,\ a]$ implies $(p, x, \$) \vdash^\star (A, z, a\alpha)$.

We have shown that $(p, x, \$) \vdash^i (A, z, a\alpha)$ implies $q_0 \rightsquigarrow^\star [A \to \cdots,\ a]$ for all $i \geq 0$; and $q_0 \rightsquigarrow^i [A \to \cdots,\ a]$ implies $(p, x, \$) \vdash^\star (A, z, a\alpha)$ for all $i \geq 0$. It follows that $(p, x, \$) \vdash^\star (A, y, a\alpha)$ if and only if $q_0 \rightsquigarrow^\star [A \to \cdots,\ a]$. $\Box$