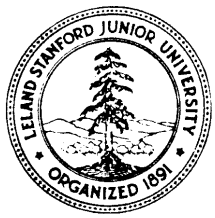
COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD, CA 943054055



AN EMPIRICAL STUDY OF AN ABSTRACT INTERPRETATION OF SCHEME PROGRAMS

Atty Kanamori
Daniel Weise

Technical Report: CSL-TR-92-520

April, 1992

This research has been supported in part by the Advanced Research Projects Agency, Department of Defense, Contract No. N0039-91-K-0138. Atty Kanamori holds an NSF Graduate Fellowship.

An Empirical Study of an Abstract Interpretation of Scheme Programs

Atty Kanamori and Daniel Weise

Technical Report: CSL-TR-92-520
(also FUSE Memo **92-11)**

April, 1992

Computer Systems Laboratory
Departments of Electrical Engineering & Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Abstract Interpretation, a powerful and general framework for performing global program analysis, is being applied to problems whose difficulty far surpasses the traditional "bit-vector" dataflow problems for which many of the high-speed abstract interpretation algorithms worked so well. Our experience has been that current methods of large scale abstract interpretation are unacceptably expensive.

We studied a typical large-scale abstract interpretation problem: computing the control flow of a higher order program. Researchers have proposed various solutions that are designed primarily to improve the *accuracy* of the analysis. The *cost* of the analyses, and its relationship to accuracy, is addressed only cursorily in the literature. Somewhat paradoxically, one can view these strategies as attempts to *simultaneously* improve the accuracy and reduce the cost. The less accurate strategies explore many spurious control paths because many *flowgraph* paths represent illegal *execution paths*. For example, the less accurate strategies violate the LIFO constraints on procedure call and return. More accurate analyses investigate fewer control paths, and therefore may be more efficient despite their increased overhead.

We empirically studied this accuracy versus efficiency tradeoff. We implemented two fixpoint algorithms, and four semantics (baseline, baseline + stack reasoning, baseline + contour reasoning, baseline + stack reasoning + contour reasoning) for a total of eight control flow analyzers. Our benchmarks test various programming constructs in isolation — hence, if a certain algorithm exhibits poor performance, the experiment also yields insight into what kind of program behavior results in that poor performance. The results suggest that strategies that increase accuracy in order to eliminate spurious paths often generate unacceptable overhead in the parts of the analysis that do not benefit from the increased accuracy. Furthermore, we found little evidence that the extra effort significantly improves the accuracy of the final result. This suggests that increasing the accuracy of the analysis globally is not a good idea, and that future research should investigate *adaptive* algorithms that use different amounts of precision on different parts of the problem.

Key Words and Phrases: Abstract Interpretation, Static Program Analysis, Control Flow Analysis

1 Introduction

Abstract Interpretation is an important and powerful tool for performing global program analysis. Traditional abstract interpretations such as live variable and reaching definition analysis of imperative programs[2] have been very practical and successful. The efficient implementation of modern programming languages, as well as the creation of parallelizing compilers for existing imperative languages, requires inventing new and expensive abstract interpretations that are much more complex than the "Dragon book" style bit-vector problems that could be solved either intraprocedurally or interprocedurally using simple, imprecise summary methods. These new abstract interpretations are interprocedural and often deal with higher order procedures and complex data structures.

For example, Harrison[6], Shivers[15], and Deutsch[4] have investigated the use of abstract interpretation on *higher order languages* i.e., languages where procedures are first class values that are dynamically created at runtime, passed to other procedures, returned from procedures, and stored in data structures. Because one cannot cheaply compute the call-graph of a higher-order program, most summary methods become useless.

Another popular application of abstract interpretation is program analysis in the presence of dynamically allocated data structures: Jones and Muchnick[7], Horwitz[5], Larus[9], Stransky[17], and Harrison[6] have applied abstract intepretation to collect information about data dependencies in the presence of pointers and data structures. These analyses compute, for each program point, a static image of the heaps that could arise just prior to executing that point. Such information is important for program parallelizers and for implementing efficient memory management.

Unfortunately, these methods, though potentially valuable, are expensive. It is not uncommon for the worse case space requirements of these methods to be cubic or even quartic with respect to the size of the program. Because the analyzers are iterative search methods that proceed monotonically towards a solution, this translates into cubic and quartic iteration counts. Because each iteration tends to become more expensive as the analysis proceeds, the actual *run* time complexity may be even worse. After all, computing an abstract heap at each program point is inherently expensive. In his Ph.D. thesis, Stransky speculates that even a highly optimized analyzer would be overwhelmed by a several thousand line program[10].

One cost lowering approach is to perform a more accurate analysis that prunes impossible flow paths. Using a more accurate, and therefore more expensive analysis may seem counter-intuitive, but the hope is that expending extra effort to compute more accurate information will pay off when the analyzer later uses this improved information to avoid analyzing spurious paths: i.e., execution paths that look like valid paths in a flowgraph sense, but, due to data dependencies, cannot occur during the execution of the flowgraph. A common example of this problem concerns procedure call and return. Many *flowgraph paths* represent illegal *execution paths* that violate the LIFO constraints on procedure call and return. Simple analyzers ignore this fact. If the savings gained due to path elimination outweigh the extra costs of computing the more accurate path information, then the more accurate analyzers will be the more efficient ones.

Despite the importance of the accuracy/efficiency tradeoffs, to the best of our knowledge, there has been no published work that systematically measured the benefits and costs of various tradeoffs in the design of abstract interpretations for large programs or for highly detailed analysis. These analyzers are fully interprocedural and often deal with higher order procedures and complex data structures rather than simple scalars. To assess the tradeoffs between accuracy and efficiency during analysis, we implemented eight different control flow analyzers using ideas developed by other workers in this field, and which follow the model of eliminating spurious analysis by improving knowledge of control flow.

Each analyzer determines for each textual call-site in a Scheme [11] program, which may be higher-order, every possible textual procedure that the call-site might call. Following Shivers [16], we refer to this analysis as *control flow* analysis, or CFA for short. We chose to study control flow analysis for three reasons. First, knowing the control flow is a prerequisite for many traditional interprocedural data flow methods, so a method of computing one for a higher order program is valuable. Olin Shivers [14, 15] addresses this problem. Second, computing control flow in a higher order language is an instance of constant folding analysis, and is therefore as difficult as any problem of computing variable ranges. First class procedures

often travel through variables and heap-allocated data structures; to track each procedure, the analyzer must compute an abstract heap at every program point. Thus, CFA does more than just compute control flow, it also computes range information for every variable and heap cell in memory. Third, when a higher order program is being analyzed, we believe that collecting precise information about *data that affect the flow of control* is of paramount importance in improving, not only the accuracy, but also the efficiency, of the analysis. When an analyzer does not have a precomputed control graph, it must use information computed *during* the analysis to determine those flowgraph paths that actually represent possible execution paths. The better it collects control data during analysis; the less time it spends chasing paths that could never occur in actual execution, and the less misinformation it collects from pursuing those paths.

The control flow analyzers differ in their abstract semantics and fixpoint algorithms. The purpose of our study was to measure the relative *costs* and *benefits* of these analyzers. We implemented two fixpoint algorithms, and four semantics (baseline, baseline + stack reasoning, baseline + contour reasoning, baseline + stack reasoning + contour reasoning) for a total of eight control flow analyzers. The baseline analyzer (CFA-BASIC) just tracks the creation of procedures, their passage through variables and data structures, and their application points. The baseline semantics examines many spurious control flow paths that could never arise at runtime. The stack semantics (CFA-STACK) removes some of the spurious paths by modeling procedure call and return. The contour semantics (CFA-CONTOUR) removes spurious paths by making finer distinctions among procedures. The fourth semantics (CFA-COMBINED) is the combination of CFA-STACK and CFA-CONTOUR.

We executed each analyzer over a set of benchmarks and recorded the cost of the analysis where the cost is defined as the number of program points processed. Our benchmarks test various programming constructs in isolation — hence, if a certain algorithm exhibits poor performance, the experiment also yields insight into what kind of program behavior results in that poor performance. Our definition of cost abstracts away machine and implementation details. Although this definition neglects the important phenomenon that the cost of processing a program point grows as the analysis proceeds, we obtain valuable insight from simply counting the program points.

The results suggest that, at least for the strategies that we have examined, increasing accuracy to eliminate spurious paths is not a good strategy. There are instances where the increased accuracy results in faster analysis (particulary for higher order programs), but overall, analysis slows down because of overhead in the parts of the program that do not benefit from the increased accuracy. Furthermore, we found little evidence that the extra effort yields significant improvements in the accuracy of the final result. This suggests that globally increasing the precision of the analysis is not a good idea. However, the savings *do* outweigh the overhead in certain cases and it appears that more work should be done investigating *adaptive* algorithms that use different amounts of precision on different parts of the program.

This paper has five more sections. The next section introduces notation and the intermediate form over which our analyzers operate. It also motivates the particular information that our analysis collects. Section 3 describes the two different fixed-point algorithms we investigated (the *memo algorithm* and the *worklist algorithm).* The fourth section presents the four semantics (the greek for the semantics appears in the Appendices). The benchmark programs and the cost/benefit interpretation of the analysis results appear in Section 5. The final section draws conclusions and and dicusses ideas for future research.

2 Preliminaries

CP S- Scheme: **A Flowgraph Language**

The eight analyzers operate on Scheme programs that have been converted into *Continuation Passing Style (CPS)* style. CPS conversion turns expression-oriented Scheme into an imperative flowgraph. That is, it transforms the nested syntax of Scheme into an equivalent program that is essentially a flowgraph disguised as a Scheme program. Figure 1 shows the formal syntax for CPS-converted Scheme[1]. CPS Scheme deliberately

[1] When **writing programs, we will follow standard Scheme notation and omit the constant., var and call tags.**

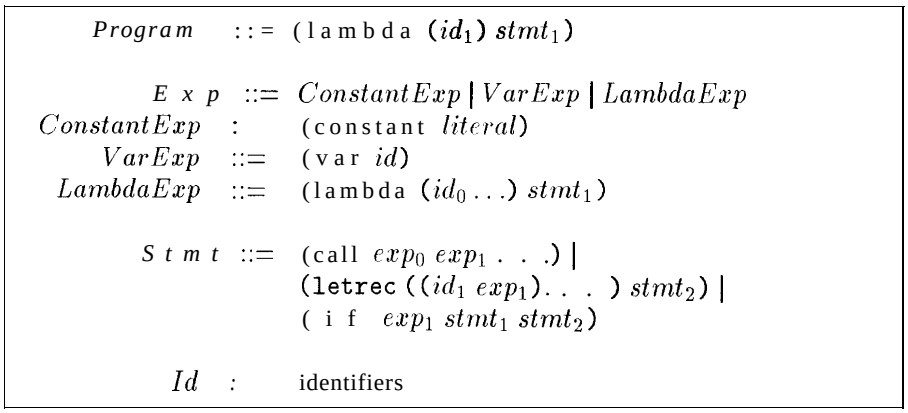


Figure 1: Formal Syntax for CPS Scheme

uses as few expression and "stmt" types as possible. All other control structures are transformed away by a macro preprocessor. Every *stmt* can be executed in a two step process: first, perform some primitive modification of the global machine state *(without* recursively executing another *stmt),* then do a `goto` to some successor *stmt.* This property makes *stmts* behave like nodes in a flowgraph, and henceforth we will use the terms "node" and "statement" interchangably. Although continuations are in every way ordinary procedures, they are inserted by the CPS converter and thus obey certain invariants. For example, in our system, continuations are always created and invoked in LIFO order. (Where necessary, we will distinguish between compiler introduced lambda expressions, `clambda`s, and user introduced lambda expressions, ulambdas.)

Notation

We assume the reader is familiar with the general theory of abstract interpretation. The uninitiated will find introductions to abstract interpretation in [1, 2, 3, 12].

Basic Objects

We use descriptive names rather than single letters to denote mathematical objects. The notation $\langle a_1, \ldots, a_n \rangle$ denotes an ordered tuple. If A is a set, 2^A represents the *power-set* of *A,* i.e., the set of all subsets of *A.* If *A* and *B* are sets, *A* x *B* represents Cartesian product. If *A* and *B* are sets, $A \rightarrow B$ denotes the set of functions that map objects from *A* to objects from *B.* If F is a function, F(a) denotes the application of F to a. This departs from the usual lambda calculus notation because we use entire words rather than single letters as identifiers. The notation $F(a_1, a_2)$ is shorthand for $(F(a_1))(a_2)$. The notation $\lambda x.E$ denotes a function. All functions are curried.

Let F be a function. The notation $F[a \mapsto b]$ denotes the function that is everywhere identical to F except that it maps a to b. If $\sqcup$ is a binary operator, the notation $F[a \mapsto_{\sqcup} b]$ denotes the function that is everywhere identical to F except that it maps a to F(n) $\sqcup$ b. The notation "if b then e_1 else e_2" is a conditional expression: if the value of b is true, the value of the conditional is e_1 otherwise, the value of the conditional is e_2.

Domains

We represent programs as flowgraphs where each node represents a basic unit of computation. We model abstract interpretation as computing a solution to a set of simultaneous equations that describe how each node modifies the information that propagates through it. The information at each node is represented by a structured value x drawn from a domain $D_{\preceq}$.

We denote domains by names subscripted by the $\preceq$ symbol. If $D_{\preceq} = \langle D, \preceq_D \rangle$ is a domain, D is the set of members of the domain and $\preceq_D$ is the partial ordering on the domain. The least upper bound operator is denoted $\sqcup_D$ and is called the *join* operator in the text. $\perp_D$ (pronounced "bottom") is the universal lower bound. We will omit the D subscript when the domain is clear from context.

We overload the $\times$, $\rightarrow$ and $\in$ notation to have meanings for domains as well as sets. That is, if $A_{\preceq} = \langle A, \preceq_A \rangle$ and $B_{\preceq} = \langle B, \preceq_B \rangle$ are domains, the notation $A_{\preceq} \times B_{\preceq}$ denotes the domain $\langle A \times B, \preceq_{A \times B} \rangle$ **where**

$$\langle a_1, b_1 \rangle \preceq_{A \times B} \langle a_2, b_2 \rangle \triangleq a_1 \preceq_A a_2 \text{ and } b_1 \preceq_B b_2$$

If S is a set and $B_{\preceq} = \langle B, \preceq_B \rangle$ is a domain, the notation $S \rightarrow B_{\preceq}$ denotes the domain $\langle S \rightarrow B, \preceq_{S \rightarrow B} \rangle$ where

$$f_1 \preceq_{S \rightarrow B} f_2 \stackrel{A}{=} \forall s \in S. f_1(s) \preceq_B f_2(s)$$

If $A_{\preceq} = \langle A, \preceq_A \rangle$ is a domain, then

$$s \in A_{\preceq} \triangleq s \in A$$

We will use the term "lattice" as a synonym for "domain."

Abstract Interpretation

Definition 1 *A* **flowgraph** *is a pair* $\langle Node, n_0 \rangle$ *where* $Node$ *is a set of nodes and* $n_0 \in$ Node *is the* initial node.

Remarks: We model programs as imperative-style flowgraphs in which each node performs some basic unit of computation by modifying a global machine state. The "edges" of the flowgraph are not explicitly specified. Instead, a node's "successors" are determined either by having the node explicitly name its successor or by looking up that information in the state. We encode the edges in the nodes and state because they cannot be computed in advance. (Indeed, the purpose of CFA is to compute this information.)

Definition 2 *An* **abstract semantics** *is an ordered quintuple (Prog, AbsState,* $\preceq$*,* q_0*, AExec) where Proy is a program* $\langle Node, n_0 \rangle$*,* $\langle AbsState, \preceq \rangle$ *is a domain,* $q_0 \in AbsState$ *is the* initial abstract state *and*

$$AExec : Node \rightarrow AbsState \rightarrow Node \rightarrow AbsState$$

Remarks: Members of *AbsState* are finite-sized values that encode sets of real machine states. The designer chooses the encoding in such a way that by examining the states, the analyzer can extract the information that the designer is seeking. We follow the "least fixed point" convention: i.e., $\perp$ represents the empty set of states, $\top$ represents the universal set of states, $q_1 \preceq q_2$ means that q_2 implies all of the possibilities that q_1 does plus possibly more, and the join operator $(\sqcup)$ is used to merge abstract states at flowgraph join points. *AExec* takes a node and an input abstract state and simulates the "execution" of the node on the abstract state. Since the abstract state encodes multiple states, there may be multiple successor nodes, and each might receive a different abstract state. Thus, *AExec* returns a function that maps successor nodes to modified incoming abstract states. This function is usually sparse (non-successor nodes are mapped to $\perp$.) The initial abstract state q_0 defines the initial machine state(s) in which programs are expected to start executing.

Strictly speaking, the abstract semantics is a function of the flowgraph being analyzed. Usually, however, th abstract semantics corresponding to a particular *problem* differ only in minor ways with respect to the flowgraph. The basic structure and description remain the same. We use the term "abstract semantics" to refer to both this general problem-dependent structure and a particular (flowgraph-dependent) instance. The intended meaning should be clear from the context.

Definition 3 *A* **solution** *to the abstract semantics defined in Definition 2 is defined to be the map from nodes to abstract states* Inf : *Node* $\rightarrow$ *AbsState such that* $Inf(n_0) = q_0$ *and*

$$\forall n \neq n_0 : Inf(n) = \bigsqcup_{n_{pred} \in Node} (AExec(n_{pred}, Inf(n_{pred})))(n)$$

Remark: *Inf* maps each node to an abstract state encoding the set of all possible real states that could be attained just prior to executing that node. The name *Inf* stands for *information.* For simplicity, we assume that there are no loops back to n_0 from within the program.

Definition 4 *A* **fixpoint algorithm** *is an effective procedure which takes as input, an abstract semantics and produces the solution to the abstract semantics.*

3 The Fixpoint Algorithms and their Costs

Our study employs two fixpoint algorithms: the *Memo* algorithm and the *Worklist* algorithm (Figure 2). The Memo algorithm remembers every abstract state that was attained at each node, while the Worklist algorithm remembers only the *join* of the abstract states attained each node. Since only the join is needed for the final result, the Memo algorithm may appear wasteful. However, this extra information is useful *during* the analysis — as described above, the more information the analyzer has regarding control-affecting data, the fewer paths it will explore. The experimental results will reveal whether this path reduction offsets the cost of remembering every abstract state attained at every node. (In general, it doesn't.)

3.1 The Memo Algorithm

We derived our memo algorithm from Olin Shivers' work [13]. The memo algorithm is a straightforward translation of a standard interpreter lifted to work on abstract states. The heart of this algorithm is the procedure *Exec* that calls *AExec* (the function that computes the abstraction) on a statement *stmt* and an abstract state *AbsState*, and then tail-calls itself recursively on the successors of the statement. To guarantee termination, Exec memoizes every $\langle stmt, AbsState \rangle$ pair it is called on – if it is called twice with the same arguments, the second call returns immediately. When *Exec* terminates, the *Inf* function is recovered from the memo-table by joining together the abstract states for each node.

As an important optimization, Exec counts one call as being the "same" as a previous call if the *stmt* was the same and the abstract state for the second call is $\preceq$ to the abstract state for the first call. The rationale is that since the abstract state for the first call already included all of the possibilities of the abstract state for the second call, pushing through with the second call will discover no new information.

The memo algorithm is intuitive and simple to program. Keeping each attained (abstract) state separate avoids the information loss suffered by the worklist algorithm which remembers only the *join* of the states at each node. The memo algorithm also has its disadvantages. Keeping track of every abstract state that reaches every node is a fairly memory-intensive operation. Performing memo-table lookup efficiently is in itself a non-trivial problem. (To get around these problems, Shivers, in his recently published thesis [16], uses an additional abstraction based on time-stamps.)

3.1.1 The Worst Case Complexity of the Memo Algorithm

Given a specific abstract domain *AbsState* and a flowgraph with *n* nodes, we can bound the worst case complexity, in terms of the number of times Exec is called, as follows:

Theorem 1 *Given a domain* $AbsState_{\preceq}$ *and a flowgraph containing n nodes, the memo algorithm given in Figure 2 will invoke* Exec *at most* $O(|AbsState|n^2)$ *times.*

Proof: The *Memo* set, which grows monotonically, has a maximum cardinality of $|Node \times AbsState| = n|AbsState|$. In the very worst case, the *Memo* set will reach this maximum size. From this point on, all remaining Exec calls will "hit" on the memo and terminate without spawning any further calls.

Every Exec call that misses on the memo will add at least 1 to the size of the memo. Hence, at the point when the memo reaches its maximum size, the number of "miss" calls that have occurred can be at most

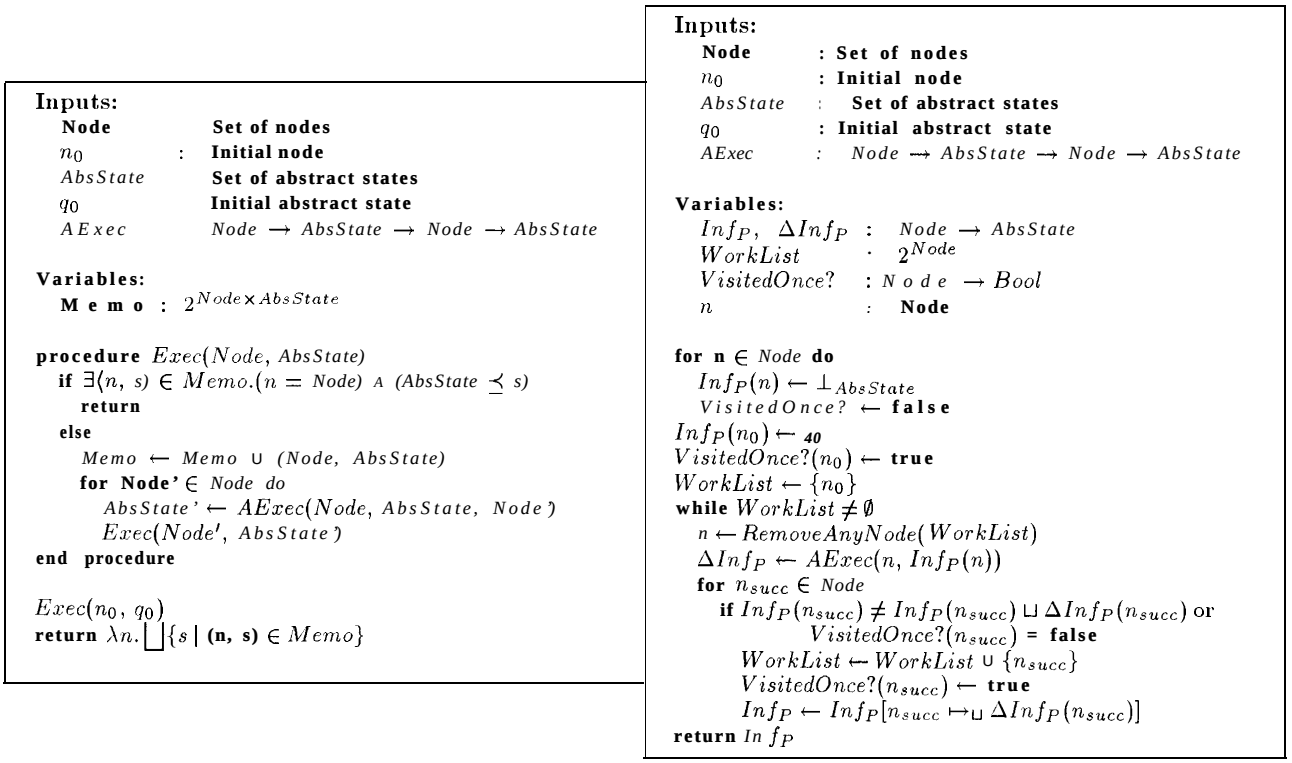


Figure 2: The Memo (Left) and Worklist (Right) Algorithms for Computing Inf_P

$|AbsState|n$. Each of these makes at most n recursive calls to Exec, hence the total number of "hit" calls that have either already occurred, or arc waiting to be unwound, is at most $|AbsState|n^2$. QED.

The size of a typical *AbsState* domain is 2^{n^k} where k ranges from 1 to 4. Thus, the worst case runtime complexity of the memo algorithm is exponential. We have only counted the number of calls to Exec, but Exec itself usually doesn't operate in constant time. Indeed, it is quite difficult to imagine a real life implementation of Exec that both operates in constant time and is as efficient as possible in the common case where the memo is relatively sparse and the abstract states are relatively close the bottom of the lattice.

3.1.2 The Average Case Complexity of the Memo Algorithm

Though the average case complexity is very problem-dependent, we can make a few useful observations by examining some of the assumptions used to derive the worst case complexity.

- Assumption: The memo set will reach its maximum size.

  This would only happen if the correct value of Inf_P was the top value of the lattice and even then, only if the algorithm were unlucky enough to generate the abstract states in precise bottom to top order. In general, the memo for each node will only contain those elements that lie below the fixpoint value of the abstract state at that node. The distance between the fixpoint value and bottom is extremely problem dependent — thus we will defer discussion of it until we describe the semantics used in our analysis.

- Assumption: The call sequence consists of a "miss" call (which then proceeds to recursively call Exec n times), followed by $n - 1$ "hit" calls which remove all but one pending call to Exec, followed by a "miss" call (which puts n more Exec's *buck* on the worklist,) followed by $n - 1$ "hit" calls, etc.

Such behavior is pathological, to put it mildly. If nothing else, a reasonably designed semantics is unlikely to be so ignorant about control flow as to propagate information to every node in the program. If one could put a constant bound k on the "fanout" of *AExec* (i.e., the number of Exec calls it spawns,) the worst case complexity of the memo algorithm would reduce to $O(k \cdot |AbsState| \cdot n) = O(|AbsState| \cdot n)$.

In an interprocedural analysis, large fanouts appear largely at return points of heavily used procedures. When we discuss the actual semantics used by our analyzers, we will provide some methods of reducing excessive fanout at return points sites.

3.2 The Worklist Algorithm

The Worklist algorithm differs from the memo algorithm in two key aspects:

- Instead of remembering every abstract state that arrived at every node, the worklist algorithm only remembers the *join* of the abstract states at each node.
- The order in which nodes are processed in no longer follows a strict depth-first predecessor-successor sequence as in the memo algorithm. Instead, any node that has information waiting to go through is put on a worklist. The algorithm is free to process nodes in the worklist in any order.

The worklist algorithm attempts to be more efficient than the memo algorithm at the cost of some precision. By maintaining only one abstract state per node, it avoids the bulky memo set of its counterpart algorithm. Given a domain $AbsState_{\prec}$ and a flowgraph containing n nodes, the worklist algorithm will invoke *AExec* at most $O(n^2 \cdot h)$ times where h is the length of the longest chain in $AbsState$.

The worklist algorithm has built-in nondeterminism. In particular, we do not specify how the helper function $RemoveAnyNode$ chooses which node to remove. The order in which nodes are processed from the worklist is left unspecified (meaning our implementation grabs whichever node is handiest, thus it does not follow an easily describable strategy). However, choosing an intelligent node order may have a significant effect on the number of iterations required to converge — the node ordering strategy represents an effecting knob to tune the performance of the worklist algorithm.

The Worst Case Complexity of the Worklist Algorithm

Theorem 2 *Given a domain $AbsState_{\prec}$ and a flowgraph containing n nodes, the worklist algorithm given in Figure 2 will invoke AExec at most $O(n^2 \cdot h)$ times where h is the length of the longest chain in $AbsState$.*

Proof: Let us classify each iteration of the worklist algorithm as "successful" or "unsuccessful ." An iteration is "successful" iff it causes *Infp* is raised to a higher value, An unsuccessful iteration occurs whenever *AExec* produces no new information to propagate — the net effect of an unsuccessful iteration is to remove one node from the worklist.

Since *Infp* grows monotonically, and is composed of n instances of $AbsState$, its height is at most $n \cdot h$. Thus, the algorithm can execute at most $n \cdot h$ successful iterations. Between each successful iteration, there can be at most $n - 1$ unsuccessful iterations. This is because each unsuccessful iteration removes a node from the worklist — the nth consecutive unsuccessful iteration would empty the worklist and terminate the algorithm. Hence, at most $h \cdot (n^2 - n)$ iterations can occur before *Infp* reaches the top of the lattice. At this point, there no further successful iterations and the algorithm will empty its worklist and terminate in at most n iterations. QED.

The Average Case Complexity of the Worklist Algorithm

The observations regarding the average case complexity of the Memo Algorithm also pertain here. First, *Infp* will not usually rise all the way to the top of the lattice. It will only rise to a least fixpoint. Second,

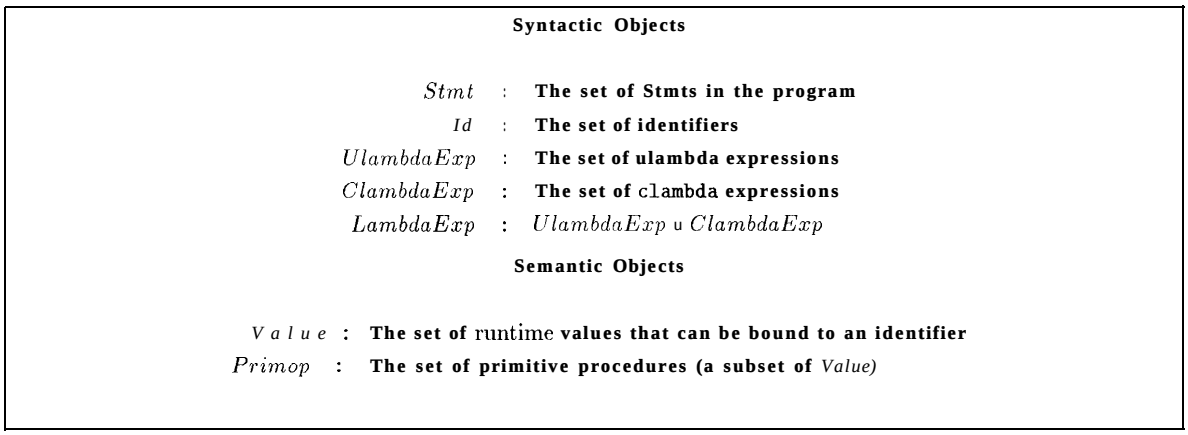


Figure 3: Concrete Sets and Domains

in a well-designed abstract interpretation, the $AExec$ function is unlikely to propagate information to every node in the program. By reasoning analogous to that for the Memo case, if one could put a constant bound k on the "fanout" of $AExec$ (i.e., the number of node it causes to be added to the worklist), the worst case complexity of the worklist would reduce to $O(k \cdot h \cdot n) = O(h \cdot n)$.

4 The Abstract Semantics

This section describes the four abstract semantics that we implemented to compute the control flow. (None of the semantics are particularly original with us, in many instances we have merely modified the Shivers' semantics [14, 15].) The four abstract semantics represent different tradeoffs of domain size versus better information about legal paths. A more detailed semantics generally corresponds to a larger domain and larger worst-case analysis time. However, the additional detail may provide the analyzer with more accurate information about control-affecting data and allow it to avoid more spurious paths, thus *decreasing* the actual analysis time.

1. **CFA-BASIC:** As its name suggests, this is a "baseline" semantics. The other three semantics are embellishments of this basic version.

2. **CFA-STACK:** This interpretation enhances CFA-BASIC by removing interprocedurally invalid execution paths from the analysis. By "interprocedurally invalid," we means paths that violate the CPS invariant that continuations are created and invoked in LIFO order.

3. **CFA-CONTOUR:** This interpretation improves the precision by separating the environments of distinct closures created from a given lambda expression.

4. **CFA-COMBINED:** Combines the enhancements of CFA-STACK and CFA-CONTOUR.

Basic syntactic and semantic objects

The semantics repeatedly refer to the sets listed in Figure 3. The syntactic objects should be straightforward. We represent statements and lambda expressions by an imaginary label that we associate with each instance. Labels are unique throughout a program. Similarly, bound identifiers are unique throughout a program (a variable renaming pass in the front end guarantees this).

CFA-STACK distinguishes between user created lambda expressions and those inserted by the CPS converter ("continuation" lambda expressions). The set $UlambdaExp$ contains the user created lambda expressions. The set $ClambdaExp$ contains the CPS inserted lambda expressions.

The set $Primop$ is a fixed set of the primitive procedures available to the Scheme programmer. This includes the special "termination continuation" that is passed to the top level ulambda.

4. 1 CFA-BASIC

Figures 10, 11, and 12 in Appendix B define CFA-BASIC. These semantics compute, at each statement, a function that maps identifiers to an *abstract value* that describes the set of values that could be bound to that identifier at runtime. The final output of the analyzer is an Inf object that maps statements to abstract states. An abstract state $AbsState$ is, for this version of the semantics, a function that maps identifiers to abstract values. The abstract values are separated into three types: $AbsPrimop$, $AbsClosure$, and $AbsBasic$. The first two represent primitive procedures and lambda procedures respectively. An abstract primop is a set of real primops. An abstract closure is, for now, the set of lambda expressions that could have produced the closure. $AbsBasic$ is a catch-all type for types such as integers and booleans. We do not provide its definition.

Interpreting CFA-BASIC with the memo algorithm will cause Exec to execute at most $O(n^2 \cdot 2^{(n^2)})$ times. Using the worklist algorithm will cause $AExec$ to execute at most $O(n^4)$ times. (The complexity calculations appear in the Appendix.)

4.2 CFA-STACK

CFA-STACK addresses an important weakness of CFA-BASIC: it does not model the LIFO order of procedure call and return. That is, a "procedure call" in CFA-BASIC returns control not only to the site that made the call but to *every* site that had called the procedure up to that point. To partially solve this problem, the CFA-STACK semantics separate different (non-recursive) calls to the same procedure. CFA-STACK introduces an abstract stack into the semantics without changing the abstract interpretation framework or the abstract state. Instead, the definition of a *node* changes from CPS-Scheme statements $(stmt)$ to $\langle stmt, stack \rangle$ pairs. Each abstract stack encodes a (possibly infinite) set of real stacks.

Problem Example

As an example of the problem of spurious multiple returns, consider a program that contains three calls to a function f (Figure 4, which shows both the program and its CPS translation). In the CPS version, the "return" from the procedure bound to f has been transformed to a call to the continuation bound to k. We notationally separate user-created lambda expressions from CPS inserted lambda expressions by using ulambdas and `clambda`s, respectively. We also number the clambda expressions for easy reference. In the discussion below, we abbreviate these clambda expressions C_1, C_2 and C_3 respectively.

Computing the fixed-point with the worklist algorithm

Consider interpreting this code (the CPS version) using the worklist algorithm. The Inf values for the body of f are initially 1. Upon encountering the first call to `f`, the worklist algorithm will propagate an abstract state (in which k is bound to {Cl}) through the body of f. Upon reaching the continuation call at the end of `f`, control will pop back out to the body of C_1 and eventually reach the second call to f.

Now, we want an abstract state (in which k is bound to $\{C_2\}$) to propagate through the body off, and want control to return to the body of C_2. Unfortunately, the worklist algorithm joins any new incoming abstract states with the abstract states previously computed for f's statements. As a result, the worklist algorithm believes that k is bound to $\{C_1, C_2\}$, and upon reaching the continuation call the second time, returns control to *both* C_1 and C_2.

Processing the third call to `f` stumbles on the same problem, only it is even worse since the effects of two previous calls to f will contaminate the third call, causing `f` to "return" to three different places on the third call. As a result, we have spawned three phantom threads of control that could not occur in the actual program. Given n different call sites, the worklist method generates on the order of n^2 spurious

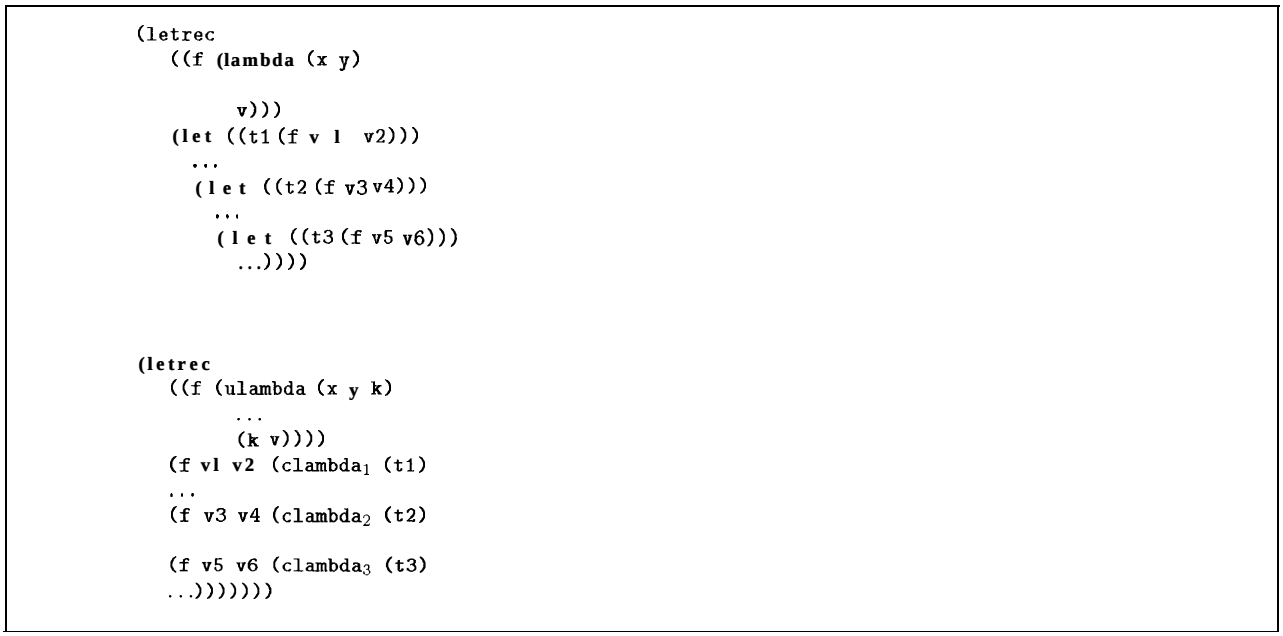


Figure 4: Demonstration For Showing CFA-BASIC weakness. The upper code is the original program, the lower code is its CPS translation.

control threads that result from procedures "returning" to the wrong place. As a result, much time is wasted "executing" these spurious threads, and additional information is lost. If this additional information loss involves procedural (or other control-affecting data), even more unnecessary paths will be pursued.

Computing the fixed-point with the memo algorithm

The memo algorithm exhibits the same problem. Although it does not join states together the way the worklist algorithm does, to simulate the binding of a variable x to a new value, it modifies the abstract state by *joining* the previous binding of x with the new value. Thus, as abstract states propagate through nodes, effects of previous (and obsolete) bindings "stick to" the abstract state. Hence, k will still be "bound" to the old clambda expressions on the second and third calls to f .

In the case of the memo algorithm, a partial solution exists. The only reason we raise, rather than overwrite, the store on binding is that one identifier may, in general, be bound to several locations simultaneously. However, in practice, many identifiers are never bound to more than one location simultaneously (global variables, for example, and formals of non-recursive procedures that are not captured by any closure that fails to die before the next call to the non-recursive procedure). A suitable prepass could identify many such variables and mark them so that for these variables, we overwrite the store rather than raising the store to simulate a binding.

Nevertheless, this solution only works for the memo algorithm and there are many cases where this does not help (recursive procedures, for example). Clearly, a better solution is needed.

4.3 Solutions

Calls to continuations implement the branch normally thought of as "procedure return." Continuations are created (i.e., their defining clambda expressions are evaluated) at the sites we normally consider as "procedure calls." Therefore continuations are created and invoked in strict LIFO order. For our CPS

converter and our input language, this invariance holds. (For the sake of brevity, we omit the proof of this[2]). The analysis should use this invariant to avoid analyzing execution paths that do not respect the LIFO sequence of continuation creation and invocation.

Sharir and Pneuli[12] proposed a solution called the "approximate call-string approach." Because it is inadequate to store return-address information in the abstract states, they proposed storing it in the nodes themselves. In simple terms, we create multiple copies of the flowgraph and tag each copy with a call-stack image. As a result, the abstract states corresponding to different calls to a procedure are kept separate and the analyzer will return to the correct point for each particular procedure activation.

This method requires no change to the existing abstract interpretation framework. It does not even require changing the abstract state. Instead, we change the definition of a *node.* Previously, nodes were CPS-Scheme statements. In this new framework, we redefine nodes to be $\langle stmt, stack \rangle$ pairs.

One problem is that the set of stack images is infinite (hence an an infinite set of nodes). Therefore, we introduce a *finite* set of abstract stacks. Each abstract stack encodes a (possibly infinite) set of real stacks. A "call-stack" in CPS-Scheme is a list of continuations waiting to be applied. In our abstract world, we represent continuations by their originating clambda's. Therefore, we encode abstract stacks as $AStack = Clambda \rightarrow$ {e, **d**, dd+} where a real stack $c_1 c_2 \ldots c_n$ is abstracted to:

$$\lambda c. \begin{cases} \texttt{e} & \text{if c appears 0 times} \\ \texttt{d} & \text{if c appears once} \\ \texttt{dd+} & \text{if c appears two or more times} \end{cases}$$

This encoding partitions the set of real stacks into equivalence classes — i.e., for every real stack, there is exactly one corresponding $Astack$. This attribute avoids propagating and storing redundant copies of the abstract state when there exist multiple copies of the flowgraph that include a given real stack in its abstract stack. Another advantage is that it "converges to a fixpoint" in at most two applications of a recursive call (an advantage over the call-string suffix approximation described by Sharir and Pneuli in which the abstract stack is a list of the last k return addresses pushed on the stack, for some arbitrary constant k — this would require the recursive call to be applied k times before fixpointing).

The CFA-STACK semantics are given in Figures 13, 14, and 15 in Appendix C. These semantics compute, for each $\langle stmt, AStack \rangle$ pair, a function that maps identifiers to an abstract value.

How does this semantics avoid the "multiple return" problems of the original? Consider Figure 4 again. Initally, $AStack$ maps every clambda to e. During the first invocation of **f**, $AStack$ maps C_1 to d and everything else to e. Upon return, C_1 is "popped off," restoring $AStack$ to its original state. Now, during the second invocation off, $AStack$ maps C_2 to **d** and everything else to e.

Now consider what happens when we reach the continuation call terminating **f** for the second time. Before, we returned to *both* the first and second continuations because k was "bound" to both of them. In CFA-STACK, it is still true that k is bound to both of them. Now, however, when $InvokeContinuation$ attempts to pop C_1 *off* the $AStack$, it will receive zero $AStacks$ back because one cannot pop a clambda that was never pushed on in the first place. Thus, it will not propagate anything back to C_1. It *will* propagate a state to C_2 because $PopContinuation$ will return a valid $AStack$ when asked to pop C_2. Thus, the $AStack$ acts to filter out (most) unwanted continuations from the overestimated information in the abstract state.

Using the memo algorithm to interpret the CFA-STACK will cause Exec to execute at most $O(n^2 \cdot 3^{2n} \cdot 2^{n^2})$ times. Using the worklist algorithm to interpret the CFA-STACK will cause $AExec$ to execute at most $O(n^4 \cdot 3^{2n})$ times. We do not have the space to provide the complete semantics for CFA-STACK, which appear in [8].

Unfortunately, our encoding does not keep track of the *order* in which continuations appear in the stack, which causes problems when analysizing recursive functions. When a CFA-STACK analyzer returns from a recursive procedure, both the "top-level" call and the "recursive" calls return to the top level, creating spurious threads and causing (potentially) an exponential blowup in the cost. This is true not only of our

[2] Note: **This means that non-local control features such as call/cc must be handled specially in the CFA-STACK framework. Handling such features is beyond the scope of this paper.**

```
(letrec
   ((compose
         (lambda (f1 f2)
             (lambda (x)
               (f1 (f2 x))))))
   (let ((add2 (compose add1 add1))
         (let ((mu14 (compose mu12 mul2)))
            ...))))
```

Figure 5: The Composition Program

semantics, but any finite semantics that models procedure call and return. Therefore, our semantics only properly models LIFO semantics for non-recursive and tail-recursive procedures.

4.4 CFA-CONTOUR

CFA-STACK and CFA-BASIC lose valuable control information when procedures are bound to user created variables because neither semantics distinguish between multiple instances of a given variable. For example, consider the "Composition" program in Figure 5, shown in non-CPS form for clarity. The procedure compose builds and returns another procedure which "carries around" the bindings of f 1 and f 2. The identifiers f 1 and f2 are bound to different values on each call to compose, and the semantics of Scheme guarantee that each closure returned by compose will independently maintain whatever values f 1 and f2 were bound to when the closure was created. Unfortunately, both CFA-BASIC and CFA-STACK allocate exactly one cell for f 1 in the abstract state that collects everything ever bound to f 1. Hence, whenever the analyzer invokes either add2 or mu14, it will branch off to both add1 and mu12 upon encountering the call to f 1. Given the importance of closures objects in functional languages, both CFA-BASIC and CFA-STACK can be expected to produce very inaccurate results for common and important programs.

The CFA-CONTOUR semantics (based on Olin Shivers' 1CFA semantics [15]) separate out closures during abstract interpretation. In an (imaginary) interpreter, every time a closure is called, the interpreter allocates a block of storage called a *contour block* to hold the closure's local variables[3]. When the code references a variable x, that reference is taken to be with respect to a certain contour block. We add a new object to the abstract machine state called a *contour map* that maps each identifier to the contour block that holds its "current" value. When a lambda expression is evaluated, the created closure captures the current contour map. When the closure is applied, this captured contour map becomes the current contour map. Whenever a closure returns a value, the original contour map is restored. However, in the CPS world, closures never return values — they pass them on to other closures -- therefore this case is moot.

In a real interpreter, there are an infinite set of contour blocks. Just as we collapsed an infinite set of stacks to a finite set to build an effective CFA-STACK analyzer, we collapse the infinite set of contour blocks to a finite set of *abstract contour blocks* to build an effective CFA-CONTOUR analyzer. (CFA-BASIC and CFA-STACK were special cases in which every contour block was merged into a single abstract contour block. Because return addresses are stored in lambda parameters like any other variables, less contour block merging would result in some, but not all, of the path removal done by stack analysis.) To achive a finite abstraction, we abstract contours as the call-sites at which they were allocated. This strategy follows naturally from the common practice of abstracting pointers as their allocation sites — since contours are merely pointers to the start of contour blocks, and contour blocks are allocated upon procedure call, it makes sense to abstract contours as call-sites.

The CFA-CONTOUR semantics are presented in Figures 17, 18 and 19 in Appendix D. Using the memo algorithm to interpret CFA-CONTOUR will cause Exec to execute at most $O(n^2 \cdot 2^{(n^4)})$ times. Using the worklist algorithm to interpret CFA-CONTOUR will cause $AExec$ to execute at most $O(n^6)$ times.

[3] **We define these as the closure's formal parameters and any variables bound by** `letrec` **statements for which the defining lambda is the immediately enclosing lambda.**

4.5 CFA-COMBINED

Our fourth abstract semantics combines the enhancements of CFA-STACK and CFA-CONTOUR. The derivation is straightforward and therefore omited for the sake of brevity.

5 The Benchmarks and Experimental Results

The benchmarks are grouped into eight *families*, each exercising a common program construct. Within each family, each benchmark is characterized by a nonnegative integer n, which is, roughly speaking, the number of times that construct appears within the benchmark. In each experiment we varied the value of n over a suitable range, used each of the eight analyzers to analyze the resulting benchmark, and recorded the number of iterations that were required for convergence. This approach highlights which program constructs give the most problems to the analyzer. It also provides insights on what kind of information is needed (and is not currently being utilized) to perform an efficient analysis.

The benchmark generators for each family appear in Figure 6. For simplicity, all of the benchmarks are shown prior to CPS conversion. Each is a "macro-expander" pattern parameterized over n that describes each benchmark in the given family. For example, the MREC pattern describes the following family of benchmarks for n being 0, 1, and 2.

```
       n=0                      n=1                          n=2
-------------------------------------------------------------------------
(let 0                  | (let 0                       | (let 0
  (define (fib n)       |   (define (fib n)           |   (define (fib n)
    (if (< n 0)         I     (if (< n 1)              |     (if (< n 2)
        n               |         n                    |         n
        (+)))           |         (+ (fib (- n 1))))) |         (+ (fib (- n 1))
                        |                              |            (fib (- n 2)))))
  (fib (read)))         I   (fib (read)))              |   (fib (read)))
```

Each benchmark was designed to exhibit a specific programming construct and nothing more. Hence, they appear quite contrived. The problem of determining how often, and in what proportion, these programming constructs appear in real-world program is a separate issue beyond the scope of this paper.

The experimental results are summarized in Figure 7 (the raw data appear in Appendix A). The "big Oh" complexity of each cost curve is provided where possible. The notations L(n) indicates a linear complexity, $Q(n)$ indicates a quadratic complexity and $C(n)$ indicates a cubic complexity. The value n is the coefficient of the leading term. A curve was declared to be L(n) only if at least six data points (at the end of the curve) can be shown to lie on a straight line. A curve was declared to be $Q(n)$ only if the difference curve (computed by taking the difference between each point and its successor) was $L(2n)$. A curve was declared to be $C(n)$ only if the difference curve of the difference curve was $L(6n)$. In some cases, extra tests were performed beyond those shown in the tables in order to meet the "six-point" criterion. For curves that do not fit polynomials, we give an ad-hoc characterization.

Multiple Recursion

A procedure is *multiply* recursive when an activation of it performs multiple top-level calls to itself. Multiple recursion occurs frequently in routines that manipulate tree structures, such as the CPS conversion pass in our analyzers. An analyzer usually executes both arms of conditionals: when the body of a procedure contains n recursive call-sites, an analyzer will believe it is **n-way** recursive, even when the call-sites are distributed among different arms of conditionals in such a way that only one or two call-sites will be reached during an actual invocation of the procedure. Procedures that manipulate abstract syntax trees inside a compiler are an example of this phenomenon. Typically, such a procedure consists of a dispatch on the type of the node it is called on.

```
(let 0
   (define (fib n)
     (if (< n n)
       n
       (+
         (fib (- n 1))

         (fib (- n n)))))
   (fib (read)))
```

MULTIPLE RECURSION (MREC)

```
(let 0
   (define (f1 x) (f2 x) (display x))

   (define (f_{n-1} x) (f_n x) (display x))
   (define (f_n x) (if x (f1 x)) (display x))
   (f1 (read)))
```

INDIRECT RECURSION (IREC)

```
(let 0
   (define (length l)
     (let loop ((l l) (n 0))
       (if (null? l)
         n
         (loop (cdr l) (+ 1 n)))))
   (length '())
   (length '(0))

   (length '(0 1 ... n - 2)))
```

LARGE FAN-IN (FAN1)

```
(let 0
   (define (length l)
     (let loop ((l l) (n 0))
       (if (null? l)
         n
         (loop (cdr l) (+ 1 n)))))
   (display_1 ... (display_n (length (read))))
   ... Repeated 8 more times. . . .
   (display_1 ... (display_n (length (read>>>>>
```

LARGE FAN-IN (FAN2)

```
(let 0
   (define (map-reduce id op f l)
     (let loop ((l l) (a id))
       (if (null? l)
         a
         (loop (cdr l) (op a (f (car l)))))))
   (define (f0 x) (+ x 0))
   ...
   (define (f_{n-1} x) (+ x n - 1))
   (map-reduce 0 + f0 '(0 1 2 3))
   ...
   (map-reduce 0 + f_{n-1} '(0 1 2 3))
```

HIGHER ORDER PROGRAMS (HO1)

```
(let 0
   (define (compose fl f2)
     (λ (x) (fl (f2 x))))
   (let
     ((f0 (compose (λ (x) (+ x 0)) (λ (y) (* y 0))))
      (f1 (compose (λ (x) (+ x 1)) (λ (y) (* y 1))))
      (f2 (compose (λ (x) (+ x 2)) (λ (y) (* y 2)))))
     ((display (f0 (read)))
      ...
      (display (f_{n-1} (read))))))
```

HIGHER. ORDER PROGRAMS (H02)

```
(let ((procs '(,+ ,- ,* ,/ ,eq?  ,eqv?)))
   (define (foo x)
     (cond
       ((eq?  x +) 0)
       ...
       ((eq?  x p_{n-1}) n - 1)
       (else n)))
   (foo (list-ref procs 0))
   (foo (list-ref procs 1))
   (foo (list-ref procs 2))
   (foo (list-ref procs 3))
   (foo (list-ref procs 4))
   (foo (list-ref procs 5)))
```

DECISION POINTS (COND1)

```
(let 0
   (define (foo x)
     (cond
       ((eq?  x +) 0)
       ((eq?  x -) 1)
       ((eq?  x *) 2)
       ((eq?  x /) 3)
       ((eq?  x eq?) 4)
       ((eq?  x eqv?) 5)
       (else 6)))
   (let loop ((ans 0)) (if (read) ans (loop (foo +))))
   ...
   (let loop ((ans 0)) (if (read) ans (loop (foo p_{n-1})))))
```

DECISION POINTS (COND2)

Figure 6: The Benchmark Families

	Memo					WorkList		
Contour / Stack	.	C .	. S	C S	. .	C .	. S	C S
MREC	Q(1)	Q(15)	EXP	>C	Q(1)	Q(1)	>C	>C
IREC	L(4)	L(4)	EXP	EXP	L(5)	L(5)	EXP	EXP
FAN1	Q(1)	C(1)	Q(4.5)	EXP	Q(0.5)	Q(0.5)	L(19)	L(19)
FAN2	L(101)	L(266)	L(65)	Insuf	L(57)	L(120)	L(20)	L(20)
HO1	Q(2.5)	>Q	>Q	EXP	Q(2)	Q(2)	Q(1)	L(65)
HO2	Q(6)	L(10)	Insuf	L(10)	Q(4)	Q(4)	Q(2)	L(10)
COND1	Q(1.5)	Insuf	Insuf	Insuf	≈L	≈L	>L	>L
COND2	C(5.2)	Insuf	Insuf	Insuf	≈Q	≈Q	L(162)	L(192)

Figure 7: Summary of Cost Complexities. EXP is apparently exponential, >C means cannot determine precisely but is worse than cubic, Insuf means Insufficient data (the test ran past the 10K limit, or was expected to, or ran out of memory), >Q means cannot determine precisely but is worse than quadratic, ≈L means Approximately linear, >L means cannot determine precisely but is worse than linear, ≈Q means Approximately quadratic.

This test is parameterized over the number of times a procedure called fib calls itself. When stack reasoning is disabled, the iteration count is quadratic with respect to n. The Memo algorithm combined with contour reasoning is significantly more expensive than the other quadratic-time analyzers. When stack reasoning is turned on, the cost increases dramatically. While we cannot precisely characterize the curves mathematically, each increase in n appears to increase the iteration count by at least a factor of 3, e.g., exponential behavior. These results appear consistent with the expected exponential behavior of stack reasoning on recursive procedures.

Indirect Recursion

A procedure P is *indirectly* recursive if it calls another procedure which in turn calls P. More than one "intermediary" procedure may be involved. This test is parameterized over the number of subprocedures that f 1 must call before reaching the recursive call to itself. In general, indirect recursion appears to cause less of a problem than multiple recursion. Nevertheless, using stack reasoning is still far more expensive than not using it.

Analyzers not using stack reasoning all exhibit linear time performance on this benchmark family. The memo algorithm performs better (smaller slope) than the worklist algorithm in this suite. It appears that, in the absence of stack reasoning, indirect recursion does not cause any special problems. The linear growth in cost is expected since the length of the path from the start of f 1 to the recursive call increases linearly with n.

When stack reasoning is turned on, however, an exponential cost curve again arises: adding one to n increases the iteration count approximately 3-fold. We pointed out previously that a sequence of calls to recursive procedures would be exponentially expensive with respect the length of the sequence. In this case, we have n recursive procedures (f 1 through f $_n$) and each recursive procedure calls the next recursive procedure. The result is a similar exponential explosion.

Large Fan-In

The *fan-in* of a lambda expression is defined as the number of call-sites from which a closure produced from that lambda expression is called. These benchmarks exercise the analyzers on lambda expressions with large fan-in. In this test, we expect CFA-STACK to outperform the other semantics. CFA-STACK imposes its own cost, however: the cost of virtually inlining the called procedure at each call-site. Hence, CFA-STACK

will help only when the cost of inlining is less than the cost of processing the extra paths created by ignoring the LIFO restriction on call-return sequences.

The benchmark family FAN1 is parameterizecl over the fan-in of the procedure length, which implements its looping behavior via tail-recursion. Because CPS conversion transforms tail-calls to gotos, this use of recursion will not create the exponential behavior for CFA-STACK discussed earlier. The performance of stack reasoning even on FAN1 is disappointing. [4] When the Memo algorithm is used without contour reasoning, the cost complexity is quadratic independent of stack reasoning, and the analysis that uses stack reasoning has a higher coefficient. When contour reasoning is used with the memo algorithm, the situation is bleaker stack reasoning transforms a cubic time algorithm into an apparently exponential time algorithm.

When the worklist algorithm is used, stack reasoning appears far more attractive: regardless of whether contour reasoning is used, stack reasoning turns a quadratic time analysis into a linear time analysis. Nonetheless, in each of the test cases, the inlining overhead of stack reasoning dominates the savings due to path elimination. With a sufficiently large fan-in, the quadratic non-stack analysis will always catch up and overtake the linear stack analysis. It appears that, even in this small example, the inlining overhead is serious.

The FAN2 benchmark family keeps the fan-in constant and varies the distance between calls to length. This varies the cost of each spurious path produced when stack effects are ignored. Not surprisingly, the algorithms that finished display linear time performance with this family.

Higher Order Programs

A higher *order* program uses procedures as full-fledged data values. The benchmark families HO1 and HO2 exercise the analyzers on higher order procedures. HO1 passes procedures into another procedure and HO2 passes them out. These families show off CFA-CONTOUR because they involve control data that lives in variables other than continuation variables (and hence, is not handled by CFA-STACK.)

The HO1 family implements a common programming construct in Scheme: a procedure that takes another procedure as an argument and applies it to a set of elements. The key line is the procedure call `(f (car l))`. The variable f is a parameter of the procedure map-reduce. The main body of the benchmark applies map-reduce to different values of f. When the analyzer cannot separate the different values of `f`, each execution of the above procedure call will cause every procedure previously assigned to f to be executed. This effect is clearly undesireable.

The HO2 family illustrates another use of higher order procedures: building functions based on other functions. The routine compose takes two procedure arguments f `1` and f `2` and returns a closure which performs the composition of the two. The compose routine may build many procedures, each of which are called from many different places in the program. It is deleterious for each such call to transfer to every f `1` and `f2` on which compose was ever called. CFA-CONTOUR partially avoids this waste by separating the storage for the different f `1` and f `2` instances. The results of the experiments using CFA-CONTOUR are mixed :

- CFA-CONTOUR is not an improvement over CFA-BASIC when the worklist algorithm is used. This is to be expected. The basic idea of contour reasoning is to separate the information regarding different instances of variables. However, some mechanism is needed to keep track of which instance is active at a particular program point. The abstract contour map is the table that maintains this information. Unfortunately, the worklist algorithm does not maintain separate contour maps for different executions of a given procedure. It only maintains the *join* of the contour maps of all previous executions of that procedure. Thus, the extra information that CFA-CONTOUR was supposed to yield is smeared out.

[4] (On this test, the raw data is adjusted to accomodate a small technical problem caused by our implementation of the analyzer. This particular benchmark program contains quoted lists. To simplify the analyzer, a prepass removes quoted lists and generates a preamble that explicitly calls the `cons` primitive and assigns the results to compiler-generated variables. In this particular example, the length of the preamble is quadratic with respect to n. Thus, *every* analyzer will display at least quadratic complexity due strictly to the preamble. This masks the effects that we are really interested in: the response of the analyzer to fan-in. Since the preamble is simply a sequential list of bindings that is executed once at the start of the benchmark, we adjust the data for its effects by subtracting the length of the preamble from each data point.

- Contour reasoning turns a quadratic cost curve into a linear cost curve when stack reasoning and the worklist algorithm is are used. The virtual inlining process *separates* the contour maps for different executions of a given procedure. Thus, the information loss associated with CFA-BASIC does not occur.

- When the memo algorithm is used, contour reasoning improves performance in the HO2 family but *hurts* performance in HO1. Indeed, HO2 is the only benchmark family tested where contour reasoning had a positive effect on cost when the memo algorithm was used. Apparently, the memo algorithm's greater sensitivity to the height of the domain overwhelms the purported advantages of CFA-CONTOUR in many cases.

Decision Points

Decision points create problems for analyzers based on abstract interpretation because, in general, analyzers must "execute" both branches of a decision. A sequence of decision points produces an exponential number of paths based on the various outcomes, which allows much potential for inefficiency. The benchmark families COND1 and COND2 display sequenced decisions. COND1 fixes the number of decision points and varies the "branching factor" (number of possible outcomes) at each decision point. COND2 fixes the branching factor and varies the number of decision points. The most striking outcome of this test is the extremely poor performance of the memo algorithm. In no instance is the memo algorithm cheaper than the corresponding worklist algorithm. When either contour or stack reasoning is activated, the cost of the memo algorithm rapidly becomes intractable.

This poor performance is not too surprising, given that the memo algorithm completely analyzes each branch before proceeding to the next. Because a flowgraph that branches at one point usually rejoins at a later point, it is desirable to analyze each of the branch paths *before* the join point before analyzing any of the nodes after the join point: if one is going to go to analyze everything after the join point, one should wait until the information collected at the join point is as complete as possible. Otherwise, the part after the join point will be reanalyzed each time new information arrives. Unfortunately, the memo algorithm's control policy outlaws a waiting strategy.

The worklist algorithm does not specify a control policy. Our specific implementation apparently (by chance) processes nodes in a reasonably efficient order: the worklist algorithm exhibits none of the blowup of the memo algorithm. This experience contains the germ of a paradigm for improving the efficiency of abstract interpretation. In this specific case, a good policy appears to be: "Do not continue analyzing past the join point until you have finished analyzing each of the branch paths." Generalizing this to a more general node ordering policy may be a useful research topic for improving the efficiency of abstract interpretation.

6 Observations and Conclusions

CFA-BASIC appears to be the only "robust" algorithm in terms of cost. When analyzed using the worklist algorithm, all of the benchmarks appear to run in linear or quadratic time.[5] CFA-BASIC analyzed with the memo algorithm appears to be the next best alternative — most of the tests ran in linear or quadratic time with the exception of the last conditional benchmark (which ran in cubic time when using the memo algorithm.)

CFA-STACK using the worklist algorithm yields cost reductions in most of the benchmarks that we tested. However, the current CFA-STACK analyzer has a serious problem in the presence of recursion: both the MREC and IREC benchmarks display serious loss of performance when stack reasoning is activated. We believe that unless this problem can be solved, CFA-STACK is not really a serious candidate for practical analysis. Our attempts at running CFA-STACK using the Memo algorithm were discouraging: many benchmarks became so slow that only a few data points could be obtained. Despite the increased

[5] This **is only approximate for the two conditional benchmarks.**

work, CFA-STACK only yielded extra accuracy for the HO2 benchmark family. In all other families, stack reasoning had no effect on the final call-graph.

CFA-CONTOUR yields disappointing results. We did not expect it to be an improvement over CFA-BASIC when used with the worklist algorithm. Unfortuately, with only one exception, it does not appear to improve upon the Memo algorithm, either. In particular, the Memo algorithm analyzing CFA-CONTOUR behaves extremely badly when faced with decision points (as the two conditional benchmarks show). Contour reasoning appears more helpful when used to enhance the worklist algorithm running CFA-STACK. In particular, it transformed a quadratic running time into a linear running time on the two Higher Order benchmarks and did not greatly increase the cost of analysis on the other benchmarks. The gain in accuracy from using contour analysis is minimal. Like CFA-STACK, CFA-CONTOUR only improved the accuracy of the HO2 benchmark.

The results favor the smaller, less accurate domains/algorithms in terms of robustness. Although CFA-CONTOUR and CFA-STACK both have the goal of reducing the analysis cost by improving the accuracy of the analysis, they both display awkward side effects that cause them to be less efficient when faced with certain kinds of problems. In particular, CFA-STACK has a serious Achilles' Heel in the area of recursion. In a similar vein, the Memo algorithm, which takes the effort to be more accurate than the Worklist algorithm, imposes a heavy cost that is not paid back by extra accuracy in most cases. It also feels unrobust in general with respect to domain enhancements: adding either contour or stack reasoning greatly increased the cost of analysis in most cases.

In certain cases, the benefits of the improved are important. For example, the second Higher Order benchmark family demonstrates an instance where Stack and Contour reasoning simultaneously improve the analysis cost and the accuracy of the final result. Such a benefit should not be given up easily.

Alternatives to global changes to the domains exist. Since the larger domains do improve the cost in the presence of certain programming constructs, but impose an overall overhead on the analysis, one solution may be to use an "adaptive" algorithm that applies the more expensive analyses (such as contour and stack reasoning) only on the parts of the problem that benefit from it. For example, CFA-CONTOUR currently creates n copies of every variables (where n is the number of call-sites.) For many problems, this is overkill. Since we are mostly interested in variables that hold control data, an alternative solution might be to use a hybrid of CFA-BASIC and CFA-CONTOUR where only the variables that hold procedures at some point in the analysis are split. Initially, the analysis proceeds like CFA-BASIC. At some point, variable x is seen to be bound to a procedure. Upon seeing this, the analyzer redefines the lattice so that x is now analyzed as in CFA-CONTOUR (i.e., the values for each instance kept separate.)

What precisely is meant by "control data."? We've been using the term informally, and in the example above, it meant "any data that is of type procedure." Clearly, this is not the ultimate definition. Even something as innocuous as the integer "6" could potentially be an index into an array of procedures and thus be considered "control data."

Nevertheless, an analyzer that dynamically modifies the lattice as it discovers which parts of the problem require extra attention could potentially alleviate many of the effiency problems we uncovered. Our experience indicates that larger domains do improve the accuracy and efficiency of abstract interpretation but only on certain parts of the problem. Hence, such adaptive algorithms may well be useful despite their greater complexity.

References

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, June 1987.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Language Design and Implementation*, pages 238-252, January 1977.

[4] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *17th Annual ACM Symposium on Principles of Programming Languages*, pages 157-168, 1990.

[5] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[6] Williams L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation: An International Journal 2:3/4:*, pages 179-396, 1989.

[7] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In *6th Annual ACM Symposium on Principles of Programming Languages*, pages 244-256, 1979.

[8] Atty Kanamori. An empirical study of an abstract interpretation of scheme programs. Unpublished Manuscript, 40 pages, July 1991.

[9] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 21-34, June 1988.

[10] Phil Pfeiffer. Unpublished synopsis of [17]. October 1990.

[11] J. Rees, W. Clinger, et al. Revised revised revised report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–76, December 1986.

[12] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189-233. Prentice-Hall, Inc., 1981.

[13] Olin Shivers. Private correspondence.

[14] Olin Shivers. Control flow analysis in Scheme. In *Conference on Programming Language Design and Implementation*, pages 1644174, June 1988.

[15] Olin Shivers. The semantics of Scheme control flow analysis (preliminary). Technical Report ERGO-90-090, CMU School of Computer Science, November 1988.

[16] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages, or, Taming Lambda.* PhD thesis, CMU School of Computer Science, May 1991.

[17] Jan Stransky. *Annlyse skmantique de structures de données dynamiques avec application au cas particulier de langages LISPiens.* PhD thesis, Université de Paris-Sud, Cent re d'Orsay, 1988.

A Raw Data

Each row corresponds to a value of n. Each cell contains either the number of iterations or one of the following. **OVER:** The analysis ran past the cutoff point of 10,000 iterations. **MEM:** The analysis overran our implementation's heap before converging. **—:** The analysis was not performed because it was decided that the likelihood termination in a reasonable time was too low to justify running the experiment.

	Memo				WorkList			
Contour / Stack	. .	C .	. S	C S	. .	C	. S	C S
0	9	9	9	9	9	9	9	
1	17	25	25	28	19	27	28	30
2	31	83	111	223	40	72	164	292
3	43	168	345	898	55	104	665	1607
4	57	282	1119	MEM	72	131	2426	7324
5	73	426	MEM		91	160	8471	OVER
6	91	600	—	—	112	191	OVER	—
COM	Q(1)	Q(15)	≈EXP	>C	Q(1)	Q(1)	>C	>C
Multiple Recursion (MREC)								
1	11	16	15	20	15	21	18	25
2	15	17	34	[redacted]	20	22	42	48
3	19	21	101		25	27	117	129
4	23	25	348		30	32	380	404
5	27	29	1243	1246	35	37	1307	1355
6	31	33	MEM		40	42	4534	4630
COM	L(4)	L(4)	≈EXP	≈EXP	L(5)	L(5)	≈EXP	≈EXP
Indirect Recursion (IREC)								
1	16	16	16	16	17	17	17	17
2	31	46	46	46	32	32	36	36
3	48	94	88	106	34	44	55	55
4	67	164	139	226	46	57	74	74
5	88	259	199	466	59	71	93	93
6	111	385	268	946	73	86	112	112
7	136	548	346	1906	88	102	131	131
8	163	754	433		104	119	150	150
9	192	1009			121	137	169	169
10	223	1319			139	156	188	188
COM	Q(1)	C (1)	Q(4.5)	≈EXP	Q(0.5)	Q(0.5)	L(19)	L(19)
Large Fan-In (FAN1) (Adjusted for Preamble)								
0	323	1539	611		224	393	207	207
1	424	2020	676		281	537	227	227
2	525	2286	741		338	657	247	247
3	626	2552	806		393	767	267	267
4	727	2818	871		450	887	287	287
5	828	3084	936		509	1017	307	307
COM	L(101)	L(266)	L(65)	Insuf	L(57)	L(120)	L(20)	L(20)
Large Fan-In (FAN2)								

Figure 8: Raw Empirical Data for Four Benchmark Families.

	Memo				WorkList			
Contour Stack	.	C	. S	C S	.	C	. S	C S
1	26	32	26	32	27	33	27	3 3
2	58	192	83	117	67	78	58	74
3	95	547	201	364	104	119	91	115
5	137	1124	412	1097	145	163	126	156
6	184	1926	1256 751	—	190	210	163	197
	236	2959		—	239	261	202	238
COM	Q(2.5)	>Q	>Q	≈EXP	Q(2)	Q(2)	Q(1)	L(65)
Higher Order Programs (HO I)								
1	13	13	13	3	3	13	13	13
2	49	23	63	23	40	70	27	23
3	105	33	454	33	69	121	45	33
4	175	43	—	43	106	168	67	45
5	253	53		53	151	221	93	55
6	345	63	—	63	204	284	123	63
7	449	73	—		73	265 354	157	75
COM	Q(6)	L(10)	Insuf	L (	10)	Q(4) Q(4)	Q(2)	L(10)
Higher Order Programs (HO2)								
0	171	943	128	1206	129	215	134	134
1	274	MEM	1646		150	254	170	170
2	287			—	306	563	296	375
3	307				380	864	354	644
4	338	—	—	—	454	1019	509	854
5	372	—	—		507	1276	683	1102
6	409	—	—		550	1486	872	1376
COM	Q(1.5)	Insuf	Insuf	Insuf	XL	≈L	> L	> L
Decision Points (COND1)								
Contour Stack		C	S	S	.	C	. S	C S
1	112	170	112	170	151	178	151	178
2	300	1691	453	1691	251	331	313	370
3	597	OVER	1476		330	473	475	562
4	1034		1636	—	502	772	637	754
5	1642			—	674	1018	799	946
6	2452				879	1304	961	1138
COM	C(5.2)	Insuf	Insuf	Insuf	≈Q	≈Q	L (1 6 2)	L(192)
Decision Points (COND2)								

Figure 9: Raw Empirical Data for 4 Benchmark Families

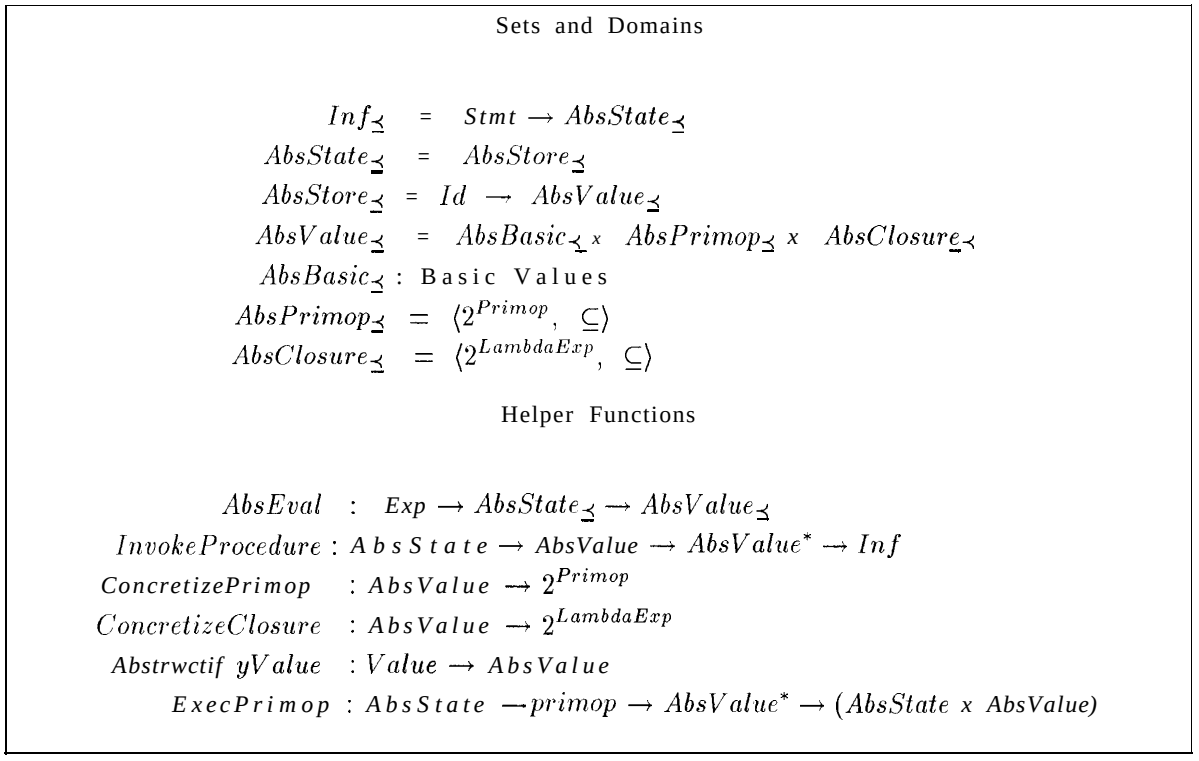


Figure 10: CFA-BASIC: Sets and Domains

B CFA-BASIC Semantics

- $AbsEval$: Simulates the action of evaluating a simple expression (constant, variable, or lambda) in a given state. This helper function only returns an abstract value since evaluating these expressions do not cause side effects.
- $InvokeProcedure$: Simulates the action of invoking a primitive procedure or a lambda closure. The first argument is the current state, the second is an abstract value representing the procedure to be called, and the third is a vector of abstract values representing the arguments. For primops, this helper function calls *ExecPrimop* to compute the result (an abstract value) and a possibly modified abstract state (modified if the primop performs side effects). Then, it passes control to $InvokeProcedure$ to pass the result to the continuation. For closures, this helper function assigns the arguments to the formals (by using join rather than overwriting since in general, each identifier could have multiple instances in the program). It leaves the "program counter" at the first statement of the called lambda.
- *ConcretizePrimop* and $ConcretizeClosure$ convert from abstract values to concrete objects. In this version of the semantics, *ConcretizePrimop* projects out the second component of the abstract value and $ConcretizeClosure$ projects out the third component.
- $AbstractifyValue$ converts a real value to its abstract counterpart. The details are not shown.
- *ExecPrimop* encapsulates the behavior of individual primops. It takes an input abstract state, a primop and a vector of abstract values (the arguments excluding the continuation) and returns a pair whose first element is a new abstract state (modified to reflect any side-effects done by the primop) and an abstract value which is the result to be passed to the continuation. The details of this helper function are not shown.

```
AbsEval(exp, AbsState) =
  let (AbsStore) = AbsState
    case exp
      [[L : (constant value)]] :AbstractifyValue( value)
      [[L : (var id)]] :          AbsStore(id)
      [[L : (lambda ...)]] :      ⟨⊥, ⊥, {L}⟩


InvokeProcedure(AbsState, AbsProcedure, AbsArgs) =
  let (AbsStore) = AbsState
      Inf_primop =
        ⨆_Inf
          {let ⟨v_1 ... v_n v_k⟩ = AbsArys
               ⟨AbsState', AbsValue⟩ = ExecPrimop(AbsState, primop, ⟨v_1 ... v_n⟩)
           InvokeProcedure(AbsState', v_k, ⟨AbsValue⟩)
          | primop ∈ ConcretizePrimop(AbsProcedure) − {#$idcont}}
      Inf_closure =
        ⨆_Inf
          { let [[(lambda (id_0 ... id_{n-1}) S: stmt)]] = LambdaExp
              if | AbsArys | ≠ n
                ⊥_Inf
                let AbsStore' = AbsStore[id_0 ↦⊔ AbsArys | 0,
                                         ...
                                         id_{n-1} ↦⊔ AbsArgs | n − 1]

                  ⊥_Inf [S ↦ (AbsStore)]
          | LambdaExp ∈ ConcretizeClosure(AbsProcedure)}
    Inf_primop ⊔ Inf_closure
```

Figure 11: CFA-BASIC: Helper Functions

```
AExec(stmt, AbsState) =
  let (AbsStore) = AbsState
    case stmt
      [[L : (call exp_0 exp_1 ... exp_n)]] :
        let AbsProcedure = AbsEval(exp_0, AbsState)
            AbsArgs = ⟨AbsEval(exp_1, AbsState) ... AbsEval(exp_n, AbsState))
            InvokeProcedure(AbsProcedure, AbsArgs)
        [[L : (letrec ((id_0 exp_0) ...) S:stmt)]]
          ⊥_Inf[S ↦ ⟨AbsStore[id_0 ↦⊔ AbsEval(exp_0, AbsState), ...]⟩]
        [[L : (if exp S_1 : stmt S_2 : stmt)]]
          ⊥_Inf[S_1 ↦ AbsState, S_2 ↦ AbsState]
```

Figure 12: CFA-BASIC: AExec

Worst Case Complexity of CFA-BASIC

The major parameters that govern the complexity of an analysis are the carclinality of the abstract state domain and the height of the longest chain in the abstract state domain. For CFA-BASIC, the height of the longest chain is $O(n^2)$, if we assume that the number of identifiers and the number of lambda expressions is proportional to n (the size of the program). The size of the domain is thus $O(2^{n^2})$. It is easy to see these results if one imagines the abstract state as being a table of bits where each row represents an identifier and each column represents a primop or lambda expression. A 1 in a given cell indicates the possibility of a certain primop or lambda being bound to the given identifier. A 0 indicates the certainty that it is not. Under our previous assumptions, the number of bits in the table is $O(n^2)$. It is also easy to verify that one abstract state is $\preceq$ to another iff the latter has a 1 bit in every cell that the former has a 1 bit. Thus, traveling up a chain corresponds to turning bits on – since there are $O(n^2)$ bits, the height, of the longest chain is $O(n^2)$. The size of the abstract state domain is the number of on-off combinations of the bits, or $O(2^{n^2})$.

Plugging these figures into our complexity formulas for the memo and worklist algorithms, we get the following results:

Theorem 3 *Using the memo algorithm to interpret CFA-BASIC will cause* Exec *to execute at most* $O(n^2 \cdot 2^{(n^2)})$ *times.*

Theorem 4 *Using the worklist algorithm to interpret CFA-BASIC will cause* $AExec$ *to execute at most* $O(n^4)$ *times.*

These results are not as gloomy as they appear. They do represent the worst case complexities — in practice, however, the average case complexities are far less than these results would suggest. In particular, the worklist algorithm generally solved the simple semantics in linear or quadratic time in most of our test runs. The memo algorithm ran in linear or quadratic time in every case except one (which ran in cubic time).

Average Case Complexity of CFA-BASIC

The average case complexity is hard to determine since it is hardly clear what an "average" program is. However, we can examine more closely our assumption that the height of the $AbsState$ lattice is really $O(n^2)$ on the average. This was derived under these assumptions:

- Assumption: The number of identifiers is proportional to the size of the program.

  This assumption seems reasonable in the average case.

- Assumption: The number of lambda expressions is proportional to the size of the program.

  This assumption also seems reasonable.

- Assumption: The number of lambda expressions that the analyzer will believe to be bound to a given identifier is proportional to the size of the program.

  We can attack this assumption on several fronts. First, even in a higher order language, many procedures are still used in a traditional first order manner — i.e., a single identifier is committed to storing that procedure throughout the program. For such identifiers, the height of the $AbsValue$ domain is effectively 1. In fact, it is not hard to design a simple prepass which will detect most of the "single-procedure" identifiers and remove them from the domains studied by the analyzer.

  More realistically, we would expect that the true height of the $AbsState$ domains is dominated by the few variables that take on a range of procedure values at runtime. Let us call these '(busy" variables. It is natural to question whether either the size of the range, or the number of busy variables, is realistically a function of the program *size* rather than of the *application*.

User-created busy variables normally take the form of procedure valued arguments to heavily used routines. Consider the built-in Scheme procedure map, for example. If map was a source-level procedure, it would have a formal parameter (let's call it f) to hold the mapping procedure. If map was called from n different places, the analyzer might iterately modify f's entry in the abstract state up to n times before reaching its final value.

Though the number of user-created busy variables is application-dependent, one would expect that in many cases, it is not proportional to the size of the program.

A more serious problem is the continuation parameters inserted by the CPS converter. Since these parameters effectively hold return addresses, they will be busy for frequently used procedures.[6] The number of such busy variables is equal to the number of user-created procedures in the source program, and thus, we would expect it to be proportional to the size of the program. The size of the ranges of such variables depends on the "fan-in" (number of call-sites) of the procedure binding the continuation. In most programs, there are a certain number of heavily used "utility" procedures whose fan-in grows as code is added to the program. Unless the *number* of such utility routines also grow proportionally with respect to the size of the program, however, it seems that the typical height of the $AbsState$ domain will not become $O(n^2)$ in general from this effect. We expect the actual figure to lie somewhere between $O(n)$ and $O(n^2)$.

- Assumption: The worst case complexity of the fixpoint algorithms are applicable.

We have stated two reasons for not believing the worst case complexities of the fixpoint algorithms. First, an actual fixpoint value is rarely the very top value in the domain. We believe that the average effective height of the CFA-BASIC $AbsState$ domain would not in general be $O(n^2)$ and could be as low as $O(n)$ under certain circumstances.

Second, the worst case analysis assumes that average fanout of $AExec$ is n — an overly pessimistic assumption. `Letrec` statements always have exactly one successor, if statements always have exactly two successors. Many call statements will have exactly one successor in practice. The trouble areas are, once again, call statements where the called procedure is computed by referencing a busy variable. If the number of such busy variables can be bounded by a constant k, the combined effects of removing this assumption and the previous one yield average case complexities of $O(n \cdot 2^n)$ and $O(n^2)$ for the memo and worklist algorithms respectively.

[6] **Although these return addresses are *represented* as "ordinary" procedure variables because of our CPS conversion, in general, it seems that a realistic analyzer would have to represent return addresses in the abstract state somehow.**

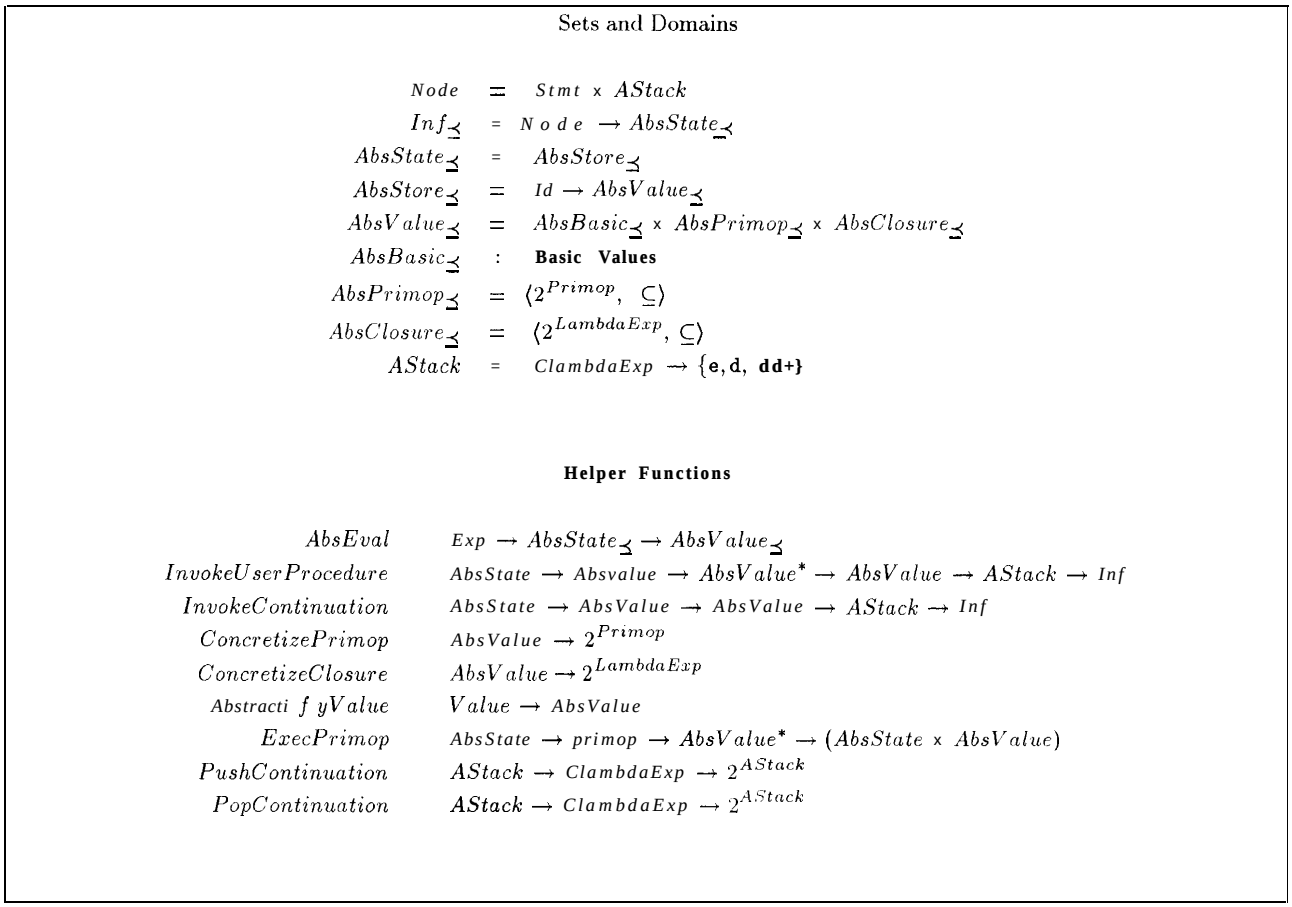


Figure 13: CFA-STACK: Sets and Domains

C CFA-STACK Semantics

These semantics compute, for each $\langle stmt, AStack \rangle$ pair, a function that maps identifiers to an abstract value.

Since CFA-STACK and CFA-BASIC are quite similar, we will only describe the differences:

- The notion of *stmt* has now been replaced by the more general notion of a *node* which is an $\langle stmt, AStack \rangle$ pair. $AExec$ now accepts a node rather than a stmt, and Inf maps nodes to abstract states.
- The new set $AStack$, as described above, is introduced.
- *Invoke Procedure* has now been replaced by two procedures: $InvokeUserProcedure$ and $InvokeContinuation$. Invoking a user procedure is different from invoking a continuation now because the latter causes the continuation to be popped off the stack. Accordingly, $InvokeContinuation$ has the stack popping code within it. $InvokeProcedure$ does *not* have any stack pushing code in it because `clambda` expressions are pushed on the stack when they are evaluated, not when their corresponding continuations are passed. The stack pushing code appears in $AExec$.
- $PushContinuation$ and $PopContinuation$ simulate the pushing and popping of continuations off the abstract stack. They are declared to return a *set* of $AStacks$ (although for this particular encoding of stacks, $PushContinuation$ never returns more than one $AStack$).

- $AExec$ now pushes a continuation on the $AStack$ whenever it evaluates a clambda expression. The `call` clause is now handled by one of four separate clauses depending on whether a user or continuation procedure is being invoked, and whether a new continuation is being created or an old one is being passed along.

The Worst Case Cost Complexity of CFA-STACK

Although the size and height of the abstract state domain are still $O(2^{n^2})$ and $O(n^2)$ from our previous results, the number of nodes has increased to $n \cdot |AStack|$. The cardinality of $AStack$ is $O(3^n)$ (3 choices of tokens for each clambda and we are assuming the number of `clambdas` is proportional to the size of the program).

Plugging these results into our worst case complexities for the algorithms we get the following:

Theorem *5 Using the memo algorithm to interpret the CFA-STACK will cause* Exec *to execute at most* $O(n^2 \cdot 3^{n} \cdot 2^{n^2})$ *times.*

Theorem *6 Using the worklist algorithm to interpret the CFA-STACK will cause* $AExec$ *to execute at most* $O(n^4 \cdot 3^{2n})$ *times.*

The Average Case Cost Complexity of CFA-STACK

Though the results derived above may appear to break new frontiers in intractability, we argue that for a certain class of programs, the average case complexity is far better than the worst case results imply.

The same arguments regarding the effective size of the abstract state domain for CFA-BASIC also hold here. However, the major cause of the complexity blowup is the exponential blowup of the node set caused by tupling the statements with abstract stacks. As Sharir and Pneuli point out, this tupling is much like inlining the code of every procedure at each call site.

In practice, however, we would only expect a tiny fraction of these nodes to be visited in an actual interpretation. How much of a fraction would this be? Let us consider the special case of a program which contains no recursion[7]. Such a program has a maximum stack length k which can be no higher than the number of plambda expressions in the program. In a program with no recursion, the token `dd+` can never appear in an abstract stack and if the maximum stack length is k, a reasonable bound on the effective number of abstract stacks is the number of abstract stacks in which up to k (out of n) entries are mapped to d. That is, the typical number of abstract stacks is

$$\sum_{i=0}^{k} \frac{n!}{(n-i)!i!}$$

If k is a fixed constant (roughly, the program has a fixed maximum call depth,) this quantity is $O(n^k)$ — in other words, polynomial rather than exponential. Caveat: this is a rough approximation, derived by computing the "big oh" complexities of each of the terms in the summation above and taking the maximum complexity. In particular, in the worse case where $k = n$, this quick rule implies that n^n abstract stacks are possible — quite a bit more than the overall 2^n limit (for nonrecursive programs).

When one introduces recursion, however, CFA-STACK reveals a serious shortcoming. When a CFA-STACK analyzer returns from a recursive procedure, both the "top-level" call and the "recursive" calls return to the top level, creating spurious threads and causing (potentially) an exponential blowup in the cost.

To illustrate the problem, consider the standard factorial program (Figure 16). The analyzer running CFA-STACK will analyze three invocations of fact. The first invocation is the one created by the top level

[7] The tail-recursive calls used to implement loops in Scheme do not count. CPS conversion effectively turns these into gotos which do not push continuations on the stack.

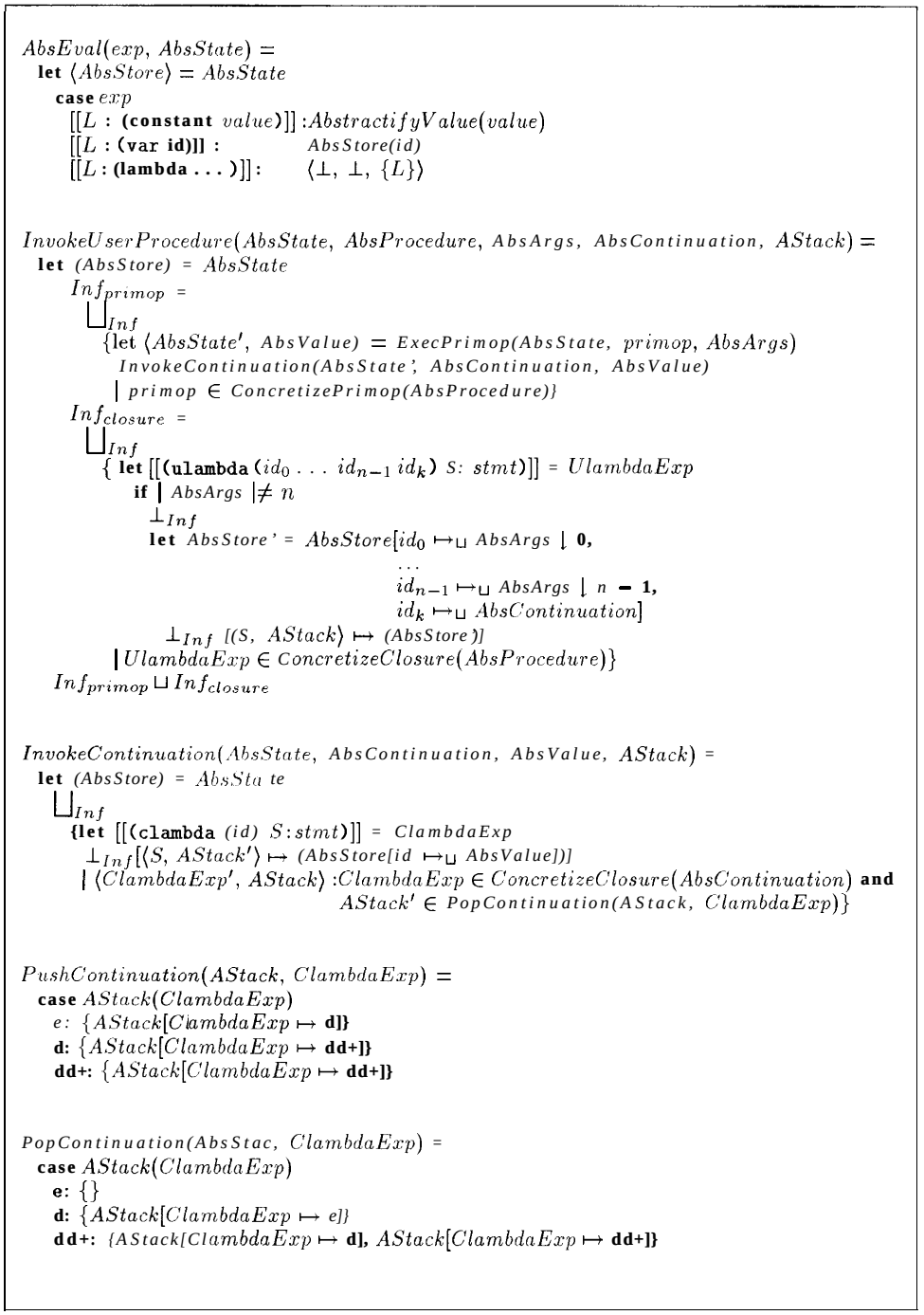


Figure 14: CFA-STACK: Helper Functions

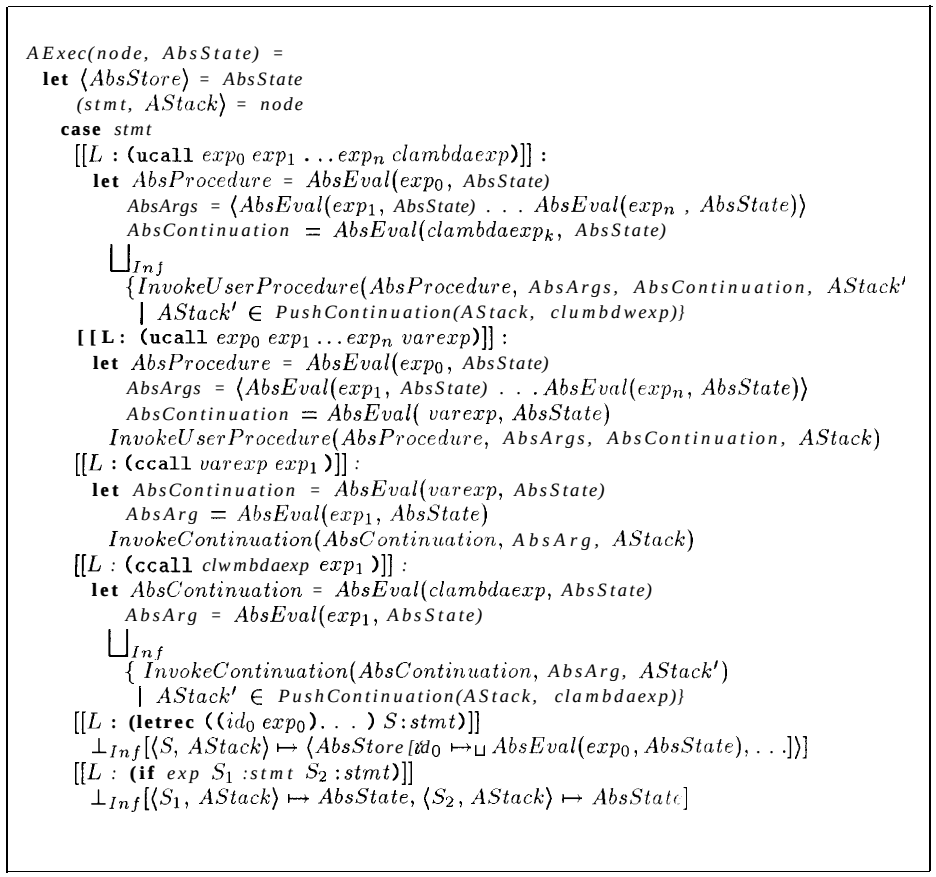


Figure 15: CFA-STACK: AExec

```
(letrec
   ((fact
         (plambda (n k)
           (= 0  n  (clambda0 (b)
           (if b
             (k 1)
             (- n  1  (clambda1 (t1)
             (fact t1 (clambda2 (t2)
             (* n t2 k))))))))))
   (fact vl (clambda3 (t1) . . .)))
```

Figure 16: The Standard Factorial Program

call to fact. Here, clambda3 will be bound to d in the abstract stack, everything else will be bound to e. The second invocation is created by the recursive call to fact. Within this invocation, clambda2 is bound to d in the abstract stack. The final invocation is also created by the recursive call to fact -- here, `clambda2` is bound to dd+ in the abstract stack.

The problem arises when one of fact's two return sites are encountered. Intuitively, the "right" thing is for only the first invocation to propagate a state back to the top level call site (clambda3). Unfortunately, a quick hand simulation shows that all three invocations "return" to the top level, passing their results to three separate copies of the flowgraph. As a result, the entire rest of the program (embodied in the clambda3 continuation) gets analyzed in triplicate. Within two of those analyses, the abstract stack is such that it implies fact recursively called itself without ever having been called from outside – an impossibility.

The problem grows worse when a procedure is multiply recursive (e.g., the naive version of Fibonacci). CFA-STACK creates 3" threads when an n-way recursive procedure is analyzed. Note that since an analyzer must usually execute both branches of a conditional, a procedure that does an **n-way** dispatch in which each branch has a single recursive call may appear to be n-way recursive to the analyzer. Tree-walking algorithms like our CPS conversion routine are a prime example of this.

A similar exponential explosion occurs when calls to recursive procedures appear in sequence. The first recursive procedure multiplies the number of execution threads by 3, the second recursive procedure multiplies each of these by 3, for a total of 9 threads.

The heart of the problem is that this particular encoding does not keep track of the *order* in which continuations appear in the stack. If the internal invocations knew, for example, that clambda3 appeared lower in the stack than clambda2, they would know not to return a state to clambda3. A stack encoding such as Sharir and Pneuli's "call-string suffix" approximation has this desirable property.

However, the call-string suffix approach has other undesirable characteristics. First, this approach will keep pushing `clambda2` on the abstract stack until the entire window is filled with `clambda2`. If the window size is w, the entire recursive procedure will be analyzed w times. One might consider detecting repeated patterns and replacing them with a special "Kleene star" construct. Unfortunately, this would make it difficult to preserve the property that $AStack$ partitions the set of real stacks into equivalence classes. The point here is that the right choice of a stack encoding is far from obvious.

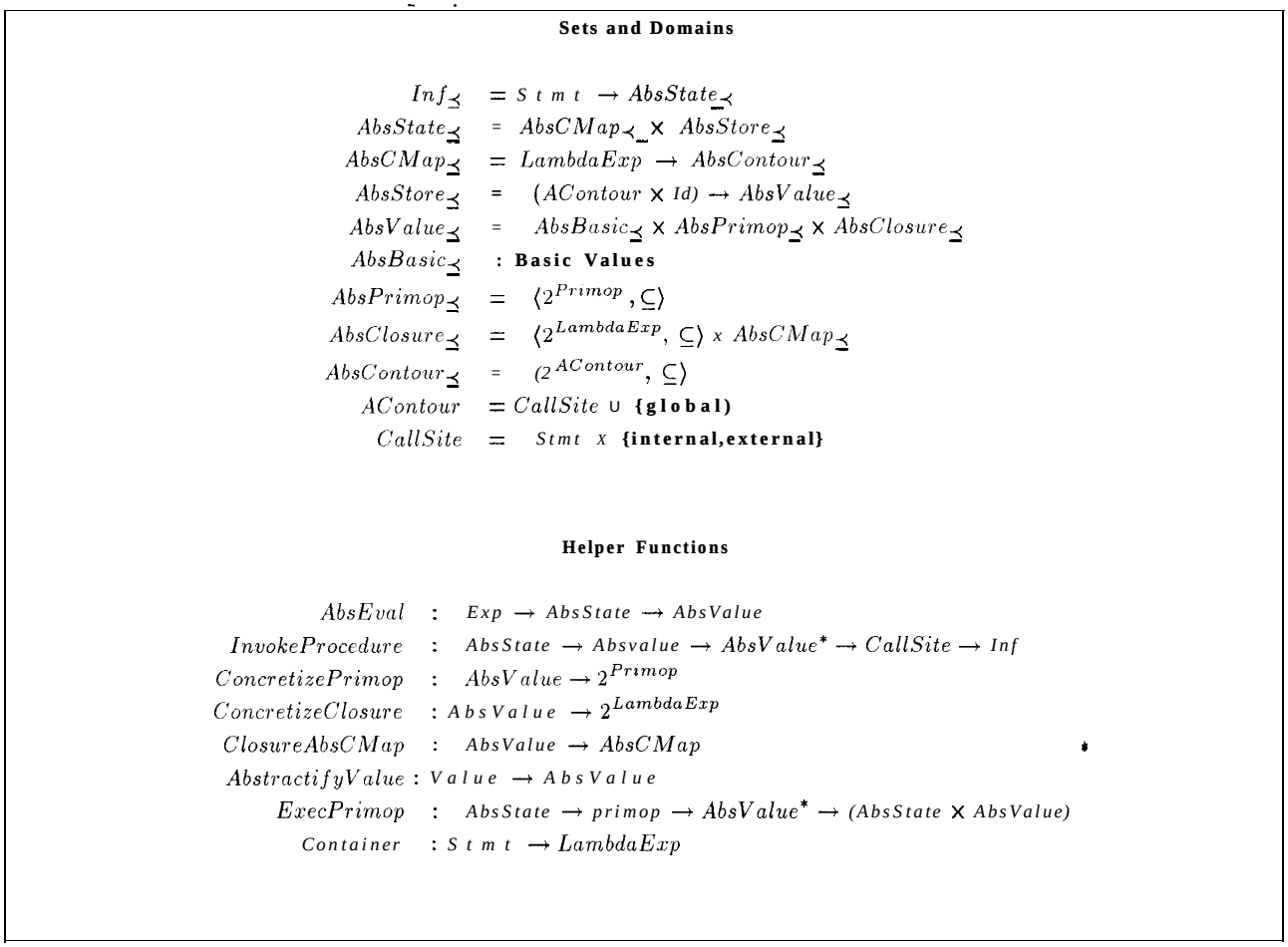


Figure 17: CFA-CONTOUR: Sets and Domains

D CFA-CONTOUR Semantics

The required changes to semantics and domains are:

- The set $AContour$ now describes the finite set of abstract contours, each of which represent a (possibly) infinite set of real contours. $AContour$s are call-sites. Call-sites come in two flavors: "external" and "internal". The "external" call-sites refer to call statements that appear explicitly in the CPS'd program. The "internal" call-sites refer to the imaginary continuation call-sites within the primops themselves. Each apply statement has its own internal call-site object to represent the internal call-site of any primops called by that statement.

  Finally, there is one special $AContour$ called global this refers the imaginary call-site that "invokes" the procedure created by the top level lambda.

- The $AbsState$ is now a pair of objects: the $AbsCMap$ and $AbsStore$. The $AbsCMap$ purportedly maps identifiers to a set of possible current $AContour$s. However, since all of the identifiers that a local to a given lambda expression live in the same contour at all times, we optimize this by having $AbsCMap$ be a function from lambda expressions to contours rather than a function from identifiers to contours. The helper function *Container* maps identifiers to the lambda expression to which they are local.

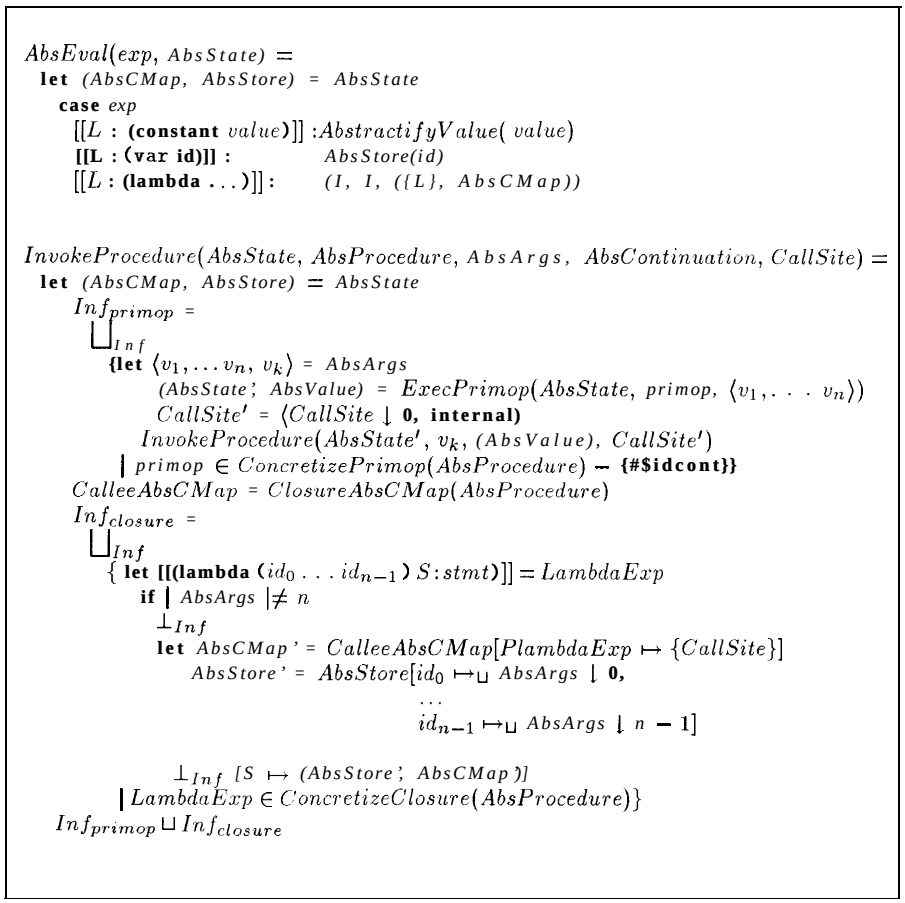


Figure 18: CFA-CONTOUR: Helper Functions

The AbsStore is similar to the original AbsState but it now maps (identifier, $acontour\rangle$ pairs to abstract values. To simulate a variable lookup, say of variable x, the analyzer first passes x to $Container$ to get the binding lambda. Then it passes this lambda to the current $AbsCMap$ to get a set of possible $AContours$. Then, for each $AContour$, it passes (x, $AContour\rangle$ to the $AbsStore$ to get the $AbsValue$ that x is bound to in that $AContour$. Finally, it joins together the $AbsValue$s for all of the $AContours$ to get the final result.

- $AbsClosures$ are now sets of lambda expressions paired with an $AbsCMap$. This reflects the fact that closures carry around their definition-time $AbsCMap$s with them.
- $AbsEval$ now captures the current $AbsCMap$ when evaluting a lambda expression.
- $Invoke\ Procedure$ and replace the current $AbsCMap$ with the called closure's definition time $AbsCMap$. They also "allocate" a new $acontour$ and update the called lambda's entry in the new $AbsCMap$.

The Worst Case Cost Complexity of CFA-CONTOUR

Computing the worst case complexity of CFA-CONTOUR is quite similar to computing the worst case complexity of CFA-BASIC. We assume that the number of identifiers, lambda expressions and call-sites are each proportional to the size of the program.

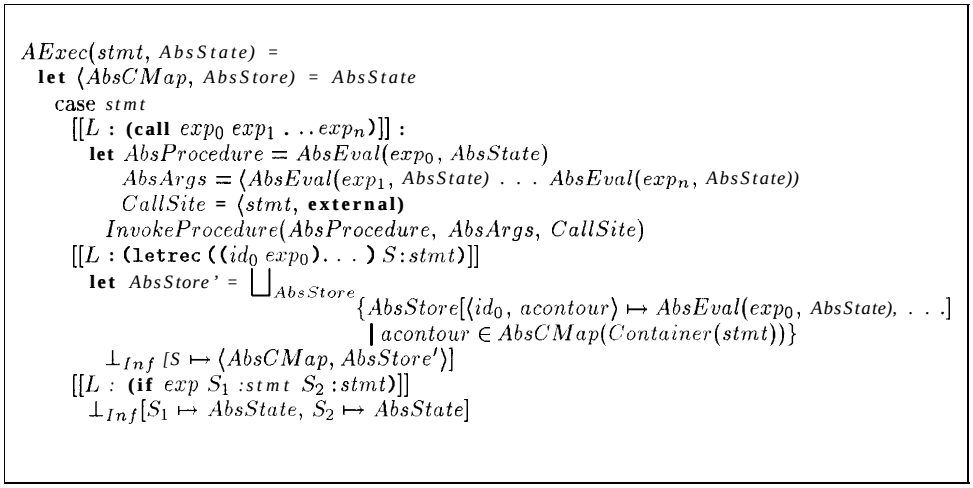


Figure 19: CFA-CONTOUR: AExec

The abstract state **now** has two components: an abstract CMap and an abstract store. Since the abstract store contains CMaps as one of its subcomponents, the latter dominates the size of the abstract state. The store contains $O(n^2)$ abstract values (one for combination of identifier and call-site). Each abstract value contains an abstract closure which contains an abstract CMap. Each abstract CMap contains O(n) abstract contours. The abstract contour domain is a powerset over call-sites, hence it has a height of O(n). Putting it all together, the height of the CFA-Contour abstract state domain is $O(n^4)$ and its size is $O(2^{(n^4)})$.

Putting these into our cost complexities for the algorithms, we get:

Theorem *7 Using the memo algorithm to interpret CFA-CONTOUR will cause* Exec *to execute at most* $O(n^2 \cdot 2^{(n^4)})$ *times.*

Theorem 8 *Using the worklist algorithm to interpret CFA-CONTOUR will cause AExec to execute at most* $O(n^6)$ *times.*

The Average Case Cost Complexity of CFA-CONTOUR,

Although CFA-CONTOUR does not introduce the exponential complexity that CFA-STACK does, $O(n^6)$ is still too large. However, we can use reasoning similar to that for C'FA-BASIC to get some handle on the average case complexity.

The number of entries in the abstract store has grown from $O(n)$ to $O(n^2)$ because of the pairing of identifiers with call-sites. However, not every lambda expression gets called from every possible call-site. Assume, for the moment, that the fan-in of every lambda expression has a constant maximum of k. Then each identifier will have at most k entries in the abstract store, restoring the number of entries to $O(n)$ status. Now, each abstract value in the store contains an abstract CMap. The height of the CMap domain is $O(n^2)$ but again, if each lambda gets called from at most k call-sites, each CMap entry will require at most k iterations to converge. Hence, the effective height of CMap in the average case is also O(n). Putting these results together, we wind up with an average effective height of $O(n^2)$.

Of course, in general, most programs do contain heavily used procedures whose fan-in is $O(n)$. If the number of such procedures is a constant, however, the linearity of the *AbsCMap* and *AbsStore* domains is preserved. While we would not assume the number of "utility" procedures to be bounded by a constant in general, this does appear to be a reasonable lower bound on the average case complexity.