Calling Context Abstraction with Shapes

Xavier Rival
INRIA Paris-Rocquencourt *
rival@di.ens.fr

Bor-Yuh Evan Chang
University of Colorado, Boulder
bec@cs.colorado.edu

Abstract

Interprocedural program analysis is often performed by computing procedure summaries. While possible, computing adequate summaries is difficult, particularly in the presence of recursive procedures. In this paper, we propose a complementary framework for interprocedural analysis based on a direct abstraction of the calling context. Specifically, our approach exploits the inductive structure of a calling context by treating it directly as a stack of activation records. We then build an abstraction based on separation logic with inductive definitions. A key element of this abstract domain is the use of parameters to refine the meaning of such call stack summaries and thus express relations across activation records and with the heap. In essence, we define an abstract interpretation-based analysis framework for recursive programs that permits a fluid per call site abstraction of the call stack—much like how shape analyzers enable a fluid per program point abstraction of the heap.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Verification

Keywords interprocedural analysis, context-sensitivity, calling context, shape analysis, inductive definitions, separation logic, symbolic abstract domain

1. Introduction

It seems there are few things, if any, more fundamental in programming than procedural abstraction. Hence, without qualification, interprocedural analysis is simply something that static program analyzers need to do well. Yet, precise interprocedural static analysis in presence of recursion is difficult—analyzers need to simultaneously abstract unbounded executions, unbounded calling context, and unbounded heap structures.

Broadly speaking, there are two main approaches to interprocedural analysis with different strengths and weaknesses. The first approach is to compute procedure summaries (e.g., [4, 16, 20]). These summaries are then used to modularly interpret function calls (i.e., derivatives of the *functional approach* [28]). This approach is exceedingly common, as modularity is important, if not a prerequisite, for scalability. Unfortunately, computing effective summaries is not easy for all families of properties. Intuitively, it is more complex to abstract relations between pairs of states than to abstract sets of states. The former is the essence of what needs to be done to compute procedure summaries, while the latter is what is more typical in program analysis.

The second approach is to perform whole program analysis that, at least conceptually, completely ignores procedural abstraction by inlining function calls. Therefore, the analysis only needs to abstract sets of states (instead of relations on pairs of states). While this kind of analysis yields context-sensitivity without the complexities of deriving procedure summaries, it is clear that the exponential blow-up makes scaling significantly more difficult. This blow-up then exerts negative pressure on the choice in precision of the state abstraction.

Certainly, these approaches are not entirely disjoint and overlap to some extent. However, in both situations, the typical result is a rather coarse abstraction of calling contexts. In the modular analysis situation, procedure summaries capture precisely what is touched by the function in question but assume the context to be arbitrary. For whole program analysis, the values of locals in the call stack above the current call are typically completely abstract. Coarse calling context abstraction makes it difficult to tackle, for example, recursive procedures where there are critical relations between successive activation records.

In this paper, we propose abstracting calling contexts precisely by directly modeling the call stack of activation records, that is, we explicitly push the call stack into the state on which we abstract. To do so, we exploit the fact that the call stack has a regular, inductive structure so that shape analysis techniques apply. Under the hood, we use separation logic formulas [21] with inductive definitions in order to elaborate precise and concise stack descriptions. We formalize our call stack abstraction inside the XISA shape analysis framework [5, 6], as we leverage this abstract domain, which is parametrized by inductive definitions.

While uncommon in practice, there has been prior work on directly abstracting the call stack [22] in the TVLA framework [27]. This approach requires careful choice of a set of predicates for the modeling of stack summaries. Instead, leveraging the natural inductive structure of the call stack and a framework built around inductive definitions, we seek to lower the configuration effort. In particular, we show that the inductive definitions for the summarization of the call stack can be derived automatically.

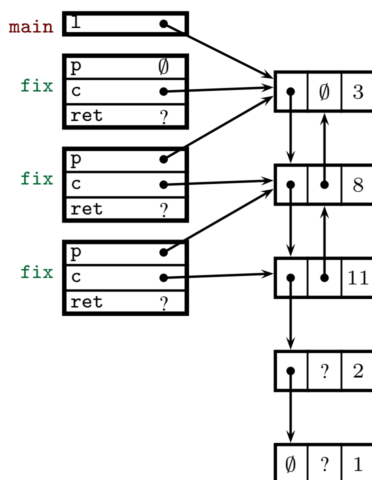
Overall, our motivation is to define global program analyses for programs with recursive functions that make use of a precise characterization of the calling context (including the status of the pending call returns). While we apply a shape domain to the abstraction of the call stack, our focus is not shape analysis per-se. Certainly, heap shapes continue to fit naturally into the framework, but there is

* Abstraction Project-team, shared with CNRS and École Normale Supérieure, Paris

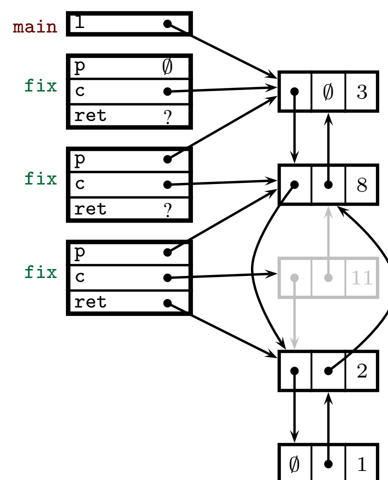
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01...$10.00

```
void main() {
  dll* l; ...
  // l is a list (maybe not doubly-linked)
  l = fix(l, NULL);
}
dll* fix(dll* c, dll* p) {
  dll* ret;
  if (c != NULL) {
    c->prev = p;
▶   c->next = fix(c->next, c);
    if (check(c->data)) {
      ret = c->next; remove(c);
▷   }
    else { ret = c; }
  }
  else { ret = NULL; }
  return ret;
}
void remove(dll* n) {
  if (n->prev != NULL)
    n->prev->next = n->next;
  if (n->next != NULL)
    n->next->prev = n->prev;
  free(n);
}
```

(a) The recursive function fix.



(b) After two recursive calls to fix and just about to make another recursive call at ▶.



(c) Before return from the second recursive call to fix at ▷.

Figure 1. An explanatory recursive program shown with diagrams depicting both the call stack and the heap at two points in its execution.

independent interest for numerical domains. In particular, the procedure summary approach has not been adapted to large classes of numerical domains, so the technique proposed in this paper takes steps towards a "plug-in" or product domain-style extension of base domains to precise interprocedural analysis.

If heap shapes are of interest, we assume our abstract domain has been instantiated with the appropriate inductive definitions describing them (e.g., they come from the user in the form of an inductive checker [6] corresponding to *structural consistency checking* code). That is, while we automatically derive a program-specific inductive definition to summarize call stacks, we do not try to infer inductive definitions for recursive heap structures. From a technical point of view, we exploit the fact that the call stack has a fixed recursive backbone—in contrast to user-defined recursive structures in the heap, which do not have a set shape. From a usability perspective, the stack of activations records is a low-level implementation mechanism for procedural abstraction and thus such inductive definitions would be problematic to expect from the user—in contrast to heap shapes, which are programmer-designed.

Finally, we clarify that we do not necessarily advocate a precise call stack abstraction in all situations. Rather our technique fills a gap in the analysis of programs with recursive procedures. In this paper, we make the following contributions:

- We define an abstract domain that models the call stack directly in an exact manner (Section 3) and with summarization (Section 4). The novel aspect of our approach is to leverage the inductive structure of the call stack by using a parametric shape domain based on separation logic and inductive definitions.
- We give an algorithm for automatically deriving inductive cases for call stack summarization (Section 5). That is, the inductive definition stack used to summarize the call stack is defined on the fly in a program-specific manner.
- We describe an analysis for programs with recursive procedures using this call stack abstraction (Section 6).
- We provide evidence through a case study that our call stack abstraction can be used to overcome precision issues in the modular approach (Section 7). That is, a less precise base domain with the call stack abstraction is sufficient for certain examples where a more precise one is needed with the modular approach.

2. Overview

In this section, we illustrate the main challenges in designing a precise abstraction of the calling context by following an example execution of the recursive function fix shown in Figure 1(a). Then, through this discussion, we preview our abstraction technique.

Consider the recursive function fix shown in Figure 1(a). This function takes as input a pointer c to a **dll** structure. A **dll** structure has three fields, next, prev, and data used to represent a doubly-linked list of integers. The function fix does two things: (1) it takes as input a singly-linked list (i.e., the prev links are unused or potentially invalid) and sets the prev field of each node to create a valid doubly-linked list; and (2) it implements a filtering operation where all nodes whose data field satisfies the check function are removed. It implements this functionality by a recursive walk using the c pointer. On each call, the p pointer points to the previous node where c->prev should be set. In particular, during the downward sequence of recursive calls, it updates c->prev to set up the doubly-linked list invariant. Then, on the upward sequence of returns, it removes all nodes whose data field satisfies check. To simplify our presentation, function fix also uses a local variable ret so that there is only one return site.

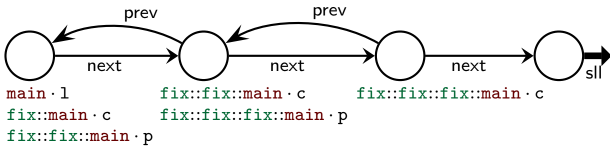
It is certainly possible to perform the filtering along the downward path of calls instead of upward path of returns, which in fact would likely make analysis easier. However, in this case, the developer has chosen to do the removal along the upward sequence, perhaps because she wants to call the library function remove. The remove function expects a doubly-linked list node, but the doubly-linked list invariant is not established until the downward sequence is complete. While this example is synthetic, it exemplifies in a

small fragment many of the key challenges in analyzing recursive imperative programs: (1) state changes on both the downward call path and upward return path, (2) incomplete or temporary breakage of data structure invariants along the recursive call-return paths, and (3) interactions with heap state conceptually "belonging" to callers.

To illustrate the analysis challenges, let us consider the concrete program state at two points in an example execution. Figure 1(b) shows the concrete state right after two recursive calls to `fix` (i.e., `main` has called `fix`, which has called itself twice and is at the program point marked with ▶). To fix a convention, we say that the first call to `fix` from `main` is not recursive. Figure 1(c) depicts the concrete state just before the return of this same call and where the current node (pointed to by `c`) has just been removed (shown grayed out). In other words, `check`(11) evaluated to true and the state is at the program point marked with ▷. We can also see that between these two states, there have been two call-returns for the last two nodes in the list (where their prev fields have been set appropriately but neither node was removed). In addition to the list allocated in the heap shown in the right part of Figures 1(b) and 1(c), we show the call stack in the left part. In our picture, the call stack consists of sequence of activation records where each field is a local variable (e.g., `p`, `c`, and `ret` for activation records of `fix`).

Our approach to interprocedural analysis is essentially to abstract the notion of state depicted in Figures 1(b) and 1(c). Traditional shape analysis focuses on precise summarization of the heap, that is, the right part in the concrete state diagrams, while the call stack is elided or coarsely abstracted. In this paper, we define a precise abstraction of the call stack—the left part in the diagrams.

To gradually build up to our technique, let us consider an intuitive abstraction of the state shown in Figure 1(b) with only heap summarization:



Here, the nodes represent heap addresses, the thin edges denote fields of a **dll** node (i.e., the edges labeled with next and prev), and the thick edge labeled sll represents a singly-linked list of **dll** nodes of undetermined length. The next and prev edges correspond to those fields of the first three nodes of the input list `l`; the data fields have been elided, as well as the prev field of the first node. The call stack is, in a sense, elided by fully qualifying variables with a call string (essentially, converting local variables into globals). For instance, the `l` variable of `main`'s activation record, `c` of the activation record for the first call to `fix`, and `p` of the activation record for the second call to `fix` all point to the first node.

Our first observation is that this kind of exact modeling of the call stack fits nicely in the separation logic-based abstraction shown above (depicted as a separating shape graph [18]) if we make the following simple extensions: (1) introduce a node for the base address of each activation record, (2) view local variables as fields of an activation record, and (3) link the activation records in a call stack together with a (conceptual) frame pointer field. Note that this representation does abstract low-level details, much like the diagram in Figure 1(b) (e.g., we do not capture contiguousness of activation records or low-level fields like the return address of each activation). This abstraction with heap summaries but an exact stack is formalized in Section 3.

At this point, we have an abstraction suitable for interprocedural analysis on non-recursive programs (but capable of precise reasoning with recursive data structures). However, for precise static analysis in the presence of recursive procedures as we propose, it is clear that we need to summarize the call stack to prevent our representation from growing unbounded. In particular, we need to abstract the concrete states both along the downward sequence of recursive calls and the upward sequence of returns. In our example, during the downward recursive call sequence, the structure of the activation records is actually quite regular: (1) the local variable `p` contains the same pointer value as `c->prev` in each activation record (even in the initial call where it is **NULL**), and (2) the next and prev fields of the already visited nodes (i.e., directly pointed to by the `c` variables in the call stack) define a valid doubly-linked list segment. It is not entirely a doubly-linked list, as `c->next` in the most recent activation record is not **NULL**. While the upward return sequence mostly preserves this pattern, there are wrinkles. In particular, in this case where a node is removed from the list, the next field of the previous element has been updated. In other words, the next field of the `c` of the second most recent activation record has been updated (i.e., `fix`::`fix`::`main` · `c` is updated when `fix`::`fix`::`fix`::`main` is still active).

Therefore, a suitable call stack abstraction must be able to precisely capture the following properties:

1. We must be able to track the *fragments* of the structure where the prev fields have been fixed. This property is needed so that we know that we obtain the doubly-linked list in the end. We also need this property to validate the call to `remove` (e.g., the prev field of the node to be removed is not dangling).
2. We must be able to track relations between the fields of each activation record and heap structures. In particular, we need to propagate invariants during the upward return sequence.

These properties can be expressed using an inductive statement since the structure of the call stack is itself inductive. Successive activation records must be disjoint regions of memory separate from each other and the heap. Thus, our second key observation is that a shape domain built on separation logic formulas with inductive definitions, such as the one described in our prior work [5, 6], seems well-suited not only for abstracting recursive heap structures precisely but also for abstracting call stacks crisply. In other words, a novel aspect of our approach is that we propose to use separation logic formulas to describe not only heap data structures but also the concrete call stack of activation records. Observe that the concrete states shown in Figures 1(b) and 1(c) are lower level than descriptions in many language semantics and most analyses.

While a shape domain based on inductive definitions seems adequate for abstracting the call stack, it is, informally speaking, necessary as well. Recall that in our example, the upward sequence of returns mostly preserves the pattern along the downward sequence of calls but not entirely. This observation indicates that the call stack abstraction must be fluid in the sense that there is a need for variation, for example, along the downward call sequence versus the upward return sequence. A similar kind of fluidity is obtained in shape analysis for heap abstraction with *materialization* [26, 27]. We observe that the classical summarization and materialization operations in shape analysis is exactly what we need at function call and function return to obtain this fluidity:

- At a function call site, the call stack grows by one activation record. We *summarize* (i.e., *fold*) the rest of call stack (excluding the new active activation record). In shape analysis, folding is done through either widening [6] or canonicalization [10, 27] operations. Folding allows us to continue the analysis with a bounded (and precise) description of the call stack. Moreover, the ability to create partial summaries is critical for addressing challenge 1 above.
- At a function return site, the compact description of the inactive activation records (i.e., the call stack excluding the most recent activation record) should be *materialized* to expose the

activation record of the caller. In shape analysis, materialization corresponds to unfolding [6, 10] or focus [27] operations. Materialization is essential for addressing challenge 2 above.

Therefore, the fundamental concepts in shape analysis provide the essential ingredients for the precise call stack abstraction that we desire. In Section 4, we describe our combined stack-heap state abstraction that makes use of an inductive predicate stack to precisely summarize recursive call stacks; the stack predicate is more complex than inductive predicates describing typical recursive data structures.

In general, shape domains require some level of parametrization to describe the structures or summaries of interest. For example, TVLA [27] uses instrumentation predicates, while separation logic-based analyzes rely on inductive definitions (either supplied by the analysis designer [2, 10] or the analysis user [6]). While asking the user to provide descriptions of user-defined structures seems quite natural, asking the user to describe the call stack does not. The call stack is not even a structure to which the user has direct access in any high-level language. However, at the same time, the inductive backbone (i.e., a stack structure) is fixed here. In Section 5, we describe a *subtraction* algorithm to automatically derive program-specific definitions of the stack predicate. Because the backbone is fixed, the challenge is in inferring the "node type" of activation records. This task is similar in spirit to Berdine et al. [2] in inferring the "node type" for a polymorphic doubly-linked list predicate.

3. Exact Call Stack Abstraction

Before we describe our approach for summarizing call contexts (see Section 4), we first formalize an exact abstraction of call stacks based on separation logic formulas.

Concrete Machine States. To begin, we describe the concrete machine states on which we abstract (see Figure 2(a)). A concrete state describes the status of a program at some point of its execution. An *environment* θ describes the set of program variables along with their machine addresses defined at a point in the execution of a program. It includes the global variables and the variables defined in each activation record in the call stack. The environment is given by the grammar in Figure 2(a): an environment encloses a stack of pairs made of a function name and a local variable to address binding before ending with a global variable binding. We write x for a generic variable drawn from $\mathbb{X}$ and use l and g to refer to local and global variable names, respectively. For any environment θ, we define a few functions to simplify our presentation. Let $\mathsf{callString}(\theta)$ denote the call string defined by θ (i.e., $p_n :: \cdots :: p_1 :: p_0$). Similarly, let $\mathsf{vars}(\theta)$ be the set of variables of environment θ. Finally, let $\mathsf{addrOf}(\theta) : \mathsf{vars}(\theta) \to \mathbb{V}$ be the function that gives the address of any variable in the environment. These functions can be defined by induction over the environment.

A *program state* s is a tuple made of an environment θ and a memory state σ. A memory state is a finite mapping from addresses to values (where the set of addresses is included in the set of values). Addresses that do not correspond to the address of any variable are *heap locations*, whereas addresses that correspond to the address of a variable defined in an activation record are *stack locations*.

3.1 Abstraction

The core of the abstraction is a spatial formula in separation logic describing the shape of memory. Spatial formulas can be seen equivalently as graphs [18]: (1) nodes, which are given symbolic names (e.g., α), abstract sets of values and (2) edges describe memory cells subject to certain constraints, such as "cell of address α contains value β" (i.e., $\alpha \mapsto \beta$). A graph G is the separating conjunction $*$ [21] of the memory regions represented by each of its

s	$::=$	$\langle \theta, \sigma \rangle$	program states ($\in \mathbb{S}$)
θ	$\in$	$\mathbb{E}$	environments
θ	$::=$	X	global variables
	$\mid$	$(p, X) :: \theta$	new activation record
σ	$\in$	$\mathbb{M} = \mathbb{V} \rightharpoonup_{\text{fin}} \mathbb{V}$	memories
X	$::=$	$\cdot \mid X, x \mapsto a$	variable to address bindings
x, l, g	$\in$	$\mathbb{X}$	program variables
p	$\in$	$\mathbb{P}$	procedure names
a	$\in$	$\mathbb{V}$	values including addresses

(a) The concrete state.

G	$::=$	$\alpha \mapsto \beta$	points-to edge
	$\mid$	$\alpha \cdot \mathsf{f} \mapsto \beta$	points-to edge of a field
	$\mid$	$\alpha \cdot \mathsf{c}(\ldots)$	inductive edge
	$\mid$	$\alpha \cdot \mathsf{c}(\ldots) \mathrel{*\!\!=} \alpha' \cdot \mathsf{c}'(\ldots)$	segment edge
	$\mid$	$\mathsf{emp} \mid G_1 * G_2$	graphs
N	$\in$	$\mathbb{D}^\sharp_{\text{num}}$	numerical constraints in a base domain
$\mathcal{A}$	$::=$	(G, N)	analysis state (i.e., abstract program state)
$\alpha, \beta, \ldots, \bar{\alpha}, g, \ldots$	$\in$	$\mathbb{V}^\sharp$	symbolic names (i.e., nodes)
$\mathsf{f}, \mathsf{g}, \ldots, \mathsf{fp}, l, \ldots$	$\in$	$\mathbb{F}$	field names (i.e., offsets)

(b) The abstract state.

Figure 2. Defining the concrete machine and abstract analysis states.

edges as shown in Figure 2(b); the empty graph is written emp. A points-to edge $\alpha \mapsto \beta$ describes a memory cell with abstract address α and contains the value abstracted by β. If we qualify the left-hand side of points-to with a field as in $\alpha \cdot \mathsf{f} \mapsto \beta$, we represent a memory cell whose address is α plus the offset of field f and whose content is β. We assume offsets are symbolic fields and thus use a relatively high-level Java-like model; a lower-level memory model could be mixed in without much difficulty by following our prior work [18]. An inductive edge $\alpha \cdot \mathsf{c}(\ldots)$ is an instance of inductive predicate c (with a distinguished traversal parameter α), while a segment edge $\alpha \cdot \mathsf{c}(\ldots) \mathrel{*\!\!=} \alpha' \cdot \mathsf{c}'(\ldots)$ is a partial derivation of an inductive predicate c. These edges summarize some set of memory cells as described by an inductive predicate allowing us to represent a potentially unbounded memory; Section 3.2 discusses these notions in greater depth. Concrete program states are then abstracted by an analysis state $\mathcal{A}$ consisting of a graph G and a numerical constraint N. A numerical constraint describes relations amongst symbolic names α and is drawn from some base domain $\mathbb{D}^\sharp_{\text{num}}$, that is, the abstract domain described is parametrized by a standard sort of numerical domain. If we instantiate this abstraction with inductive definitions describing recursive heap data structures supplied by the user, we essentially obtain the shape domains in our prior work [5, 6].

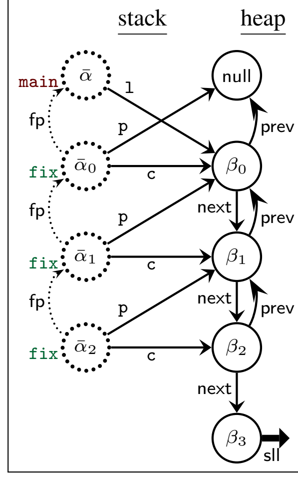
To abstract the call stack in a concrete state $s = \langle \theta, \sigma \rangle$, we observe that the environment θ plays a significant role here. At the abstract level, we build it directly into the graph as follows:

- The address of each global is represented by a node. Thus, the set of global variable names are included in the set of symbolic names $\mathbb{V}^\sharp$.
- We introduce a node to represent the base address of each activation record. For the sake of clarity, we distinguish nodes representing activation record addresses by using a bar over the symbolic names as in $\bar{\alpha}$ and by drawing them with a bold, dotted border in diagrams. We also annotate them with the function to which they correspond (i.e., indicating the "type" of the activation record).

- Each local variable can be viewed as a field of its activation record. The set of field names $\mathbb{F}$ therefore includes local variable names. This representation is key in allowing locals to be summarized as part of the call stack (see Section 4).
- Lastly, we make explicit a *frame pointer* fp, which is simply a field of all activation records. Like in the physical memory at run time, the frame pointer fp points to the previous activation record in the call stack. To distinguish them clearly in the diagrams, frame pointer edges are drawn as dotted lines (and usually the fp label is omitted).

Thus, with the above observations, we encode the structure of the call stack in our shape domain in a faithful manner essentially as-is and without any significant modifications.

As an example, an abstraction of the concrete state from Figure 1(b) is shown inset. The $\bar{\alpha}$ nodes represent the base address of the activation records, and their outgoing points-to edges correspond to the call stack. Observe that the horizontal edges between the $\bar{\alpha}$ nodes and the β nodes are the stack cells for local variables. Meanwhile, the fp links connect the $\bar{\alpha}$ nodes to capture the actual stack structure. The heap is represented by the edges in the rightmost column: the vertical edges are the next and prev fields for the first three nodes and $\beta_3 \cdot \mathsf{sll}()$ summarizes an arbitrary singly-linked list. Note that this portion of the graph is the only part of the memory state that is captured by traditional shape analyses. We have elided a few edges from this figure, specifically the `ret` variable edges in the stack and the `data` fields in the heap.



Concretization. Like for concrete states, we define functions that compute the call string $\mathsf{callString}(G)$ and the set of variables it defines $\mathsf{vars}(G)$ given a graph G. These functions follow the chain of frame pointers to compute the desired result. We also define a function

$$\mathsf{addrOf}^\sharp(G) : \mathsf{vars}(G) \to \mathbb{V}^\sharp \times \mathbb{F}$$

that maps each variable to the points-to edge representing its cell (i.e., an abstract address-of mapping).

To define the concretization of a graph, we need a mapping between symbolic names and concrete values. Such mappings $\nu : \mathbb{V}^\sharp \to \mathbb{V}$ are called *valuations* [5] and allow us to abstract irrelevant details like concrete physical addresses. We first summarize the types of the concretizations (the names of the various domains are given in the inset):

$\mathbb{D}^\sharp_{\mathrm{graph}}$	graph domain
$\mathbb{D}^\sharp_{\mathrm{num}}$	numerical domain
$\mathbb{D}^\sharp = \mathbb{D}^\sharp_{\mathrm{graph}} \times \mathbb{D}^\sharp_{\mathrm{num}}$	shape domain

$$\begin{array}{rcccl} \boldsymbol{\gamma}_{\mathrm{graph}} & : & \mathbb{D}^\sharp_{\mathrm{graph}} & \to & \mathcal{P}(\mathbb{M} \times (\mathbb{V}^\sharp \to \mathbb{V})) \\ \boldsymbol{\gamma}_{\mathrm{num}} & : & \mathbb{D}^\sharp_{\mathrm{num}} & \to & \mathcal{P}(\mathbb{V}^\sharp \to \mathbb{V}) \\ \boldsymbol{\gamma} & : & \mathbb{D}^\sharp & \to & \mathcal{P}(\mathbb{E} \times \mathbb{M}) \ . \end{array}$$

The concretization of a graph G yields a set of pairs consisting of a concrete memory σ and a valuation ν, while concretizing a numerical domain element N should give a set of valuations. Together, the concretization of the combined domain produces a set of pairs of a concrete environment θ and a concrete memory σ.

To concretize a graph, we take the concretization of each edge:

$$\begin{array}{lcl} \boldsymbol{\gamma}_{\mathrm{graph}}(\mathsf{emp}) & \stackrel{\text{def}}{=} & \{ ([\cdot], \nu) \mid \nu \in (\mathbb{V}^\sharp \to \mathbb{V}) \} \\ \boldsymbol{\gamma}_{\mathrm{graph}}(G_1 * G_2) & \stackrel{\text{def}}{=} & \{ (\sigma_1 \otimes \sigma_2, \nu) \mid (\sigma_1, \nu) \in \boldsymbol{\gamma}_{\mathrm{graph}}(G_1) \\ & & \quad \text{and } (\sigma_2, \nu) \in \boldsymbol{\gamma}_{\mathrm{graph}}(G_2) \} \ . \end{array}$$

We write $\otimes$ for the separating conjunction of concrete memories (i.e., the union of two memory maps with disjoint domains) and $[\cdot]$ for an empty concrete memory. The frame pointer fp fields are model fields, so they have no concrete correspondence, but otherwise, the concretization of a points-to edge is a single memory cell, written $[a_1 \mapsto a_2]$:

$$\begin{array}{lcl} \boldsymbol{\gamma}_{\mathrm{graph}}(\bar{\alpha}_1 \cdot \mathsf{fp} \mapsto \bar{\alpha}_2) & \stackrel{\text{def}}{=} & \{ ([\cdot], \nu) \mid \nu \in (\mathbb{V}^\sharp \to \mathbb{V}) \} \\ \boldsymbol{\gamma}_{\mathrm{graph}}(\alpha \cdot \mathsf{f} \mapsto \beta) & \stackrel{\text{def}}{=} & \{ ([\nu(\alpha, \mathsf{f}) \mapsto \nu(\beta)], \nu) \mid \\ & & \quad \nu \in (\mathbb{V}^\sharp \to \mathbb{V}) \} \\ \boldsymbol{\gamma}_{\mathrm{graph}}(\alpha \mapsto \beta) & \stackrel{\text{def}}{=} & \{ ([\nu(\alpha) \mapsto \nu(\beta)], \nu) \mid \nu \in (\mathbb{V}^\sharp \to \mathbb{V}) \} \end{array}$$

where $\nu(\alpha, \mathsf{f})$ gives the base address α plus the offset of field f. We postpone defining the concretization of summary edges (i.e., inductive and segment edges) to Section 3.2.

Overall, the concretization of an analysis state $\mathcal{A}$ are the environment-memory pairs given by the graph and consistent with the numerical constraint:

$$\begin{array}{l} \langle \theta, \sigma \rangle \in \boldsymbol{\gamma}(G, N) \quad \text{iff} \quad \text{for some } \nu, \\ \quad \mathsf{callString}(\theta) = \mathsf{callString}(G) \quad \text{and} \quad \mathsf{vars}(\theta) = \mathsf{vars}(G) \\ \quad \text{and} \quad (\sigma, \nu) \in \boldsymbol{\gamma}_{\mathrm{graph}}(G) \quad \text{and} \quad \nu \in \boldsymbol{\gamma}_{\mathrm{num}}(N) \quad \text{and} \\ \quad \mathsf{addrOf}(\theta)(x) = \nu(\mathsf{addrOf}^\sharp(G)(x)) \quad \text{for all } x \in \mathsf{vars}(\theta) \ . \end{array}$$

Valuations ν connect the various components, capturing relations across disjoint memory regions and with the numerical constraint.

3.2 Inductive Summarization and Materialization

As alluded to in Section 3.1, we summarize a potentially unbounded memory using edges built on inductive definitions. At a high-level, an inductive definition consists of a set of *unfolding rules* or cases that specify how a memory region can be recognized through a recursive traversal. As stated earlier, our inductive edges come in two forms: (1) an *inductive edge* $\alpha \cdot \mathsf{c}(\ldots)$ describes a memory region that satisfies inductive definition c from α, and (2) a *segment edge* $\alpha \cdot \mathsf{c}(\ldots) \mathrel{*\!\!=} \alpha' \cdot \mathsf{c}'(\ldots)$ describes an incomplete structure, in particular, a memory region that can be derived by unfolding $\alpha \cdot \mathsf{c}(\ldots)$ a certain number of times up to a (missing) $\alpha' \cdot \mathsf{c}'(\ldots)$ sub-region [5]. Relations among pointers or numerical values between successive unfoldings of an inductive edge are captured by definitions with additional parameters. For instance, the relation between prev and next pointers in a doubly-linked list can be captured by the following inductive definition (written as a separation logic formula);

$$\begin{array}{l} l \cdot \mathsf{dll}(p) \stackrel{\text{def}}{=} \big(\mathsf{emp} \wedge l = \mathsf{null}\big) \ \vee \ \big(\exists n, d. \\ \qquad (l \cdot \mathsf{prev} \mapsto p \ * \ l \cdot \mathsf{next} \mapsto n \ * \ l \cdot \mathsf{data} \mapsto d \\ \qquad\quad * \ n \cdot \mathsf{dll}(l)) \wedge l \neq \mathsf{null}\big) \end{array}$$

Unfolding. An inductive definition gives rise to a natural syntactic *unfolding* operation. Unfolding substitutes an inductive edge $\alpha \cdot \mathsf{c}(\ldots)$ or a segment edge $\alpha \cdot \mathsf{c}(\ldots) \mathrel{*\!\!=} \alpha' \cdot \mathsf{c}'(\ldots)$ with one inductive case of c's definition (and where all existentially-quantified variables are replaced with fresh nodes). For inductive edges, base cases correspond to a rule with no new inductive edge upon unfolding; for segment edges, the base case is unfolding to the empty segment (i.e., when $\alpha = \alpha'$ and $\mathsf{c} = \mathsf{c}'$). We write $G \rightsquigarrow_{\mathbf{unfold}} G'$ for an unfolding step from graph G to G', as well as $\rightsquigarrow^\star_{\mathbf{unfold}}$ for the reflexive-transitive closure of $\rightsquigarrow_{\mathbf{unfold}}$. Because the unfolding operation is so closely tied to the inductive definition, we often present an inductive definition by the unfoldings that it induces.

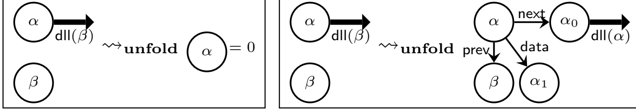


Figure 3. Unfolding operation induced by the dll definition.

For instance, in Figure 3, we present the doubly-linked list definition dll in this style.

The concretization of graphs containing inductive or segment edges is based on the concretization without them. In particular, the concretization of a graph G containing inductives is simply the join of the concretizations of all the graphs that can be derived from it by successive unfolding:

$$\gamma_{\text{graph}}(G) = \bigcup \{G' \in \mathbb{D}^{\sharp}_{\text{graph}} \mid G \rightsquigarrow^{\star}_{\textbf{unfold}} G'\} \; .$$

Unfolding and Analyzing Updates. Unfolding is the key operation for abstractly interpreting writes. To reflect an update $e_1 := e_2$, we traverse the graph to find the points-to edges (i.e., the memory cells) that correspond to the addressing expressions e_1 and e_2. If the points-to edges of interest already exist in the graph, then reflecting the update is simply a matter of swinging an edge. Because a graph is a separating conjunction of edges, this modification is a strong, destructive update. However, if the desired edges are not present, then we try to materialize them with the unfolding operation $\rightsquigarrow_{\textbf{unfold}}$. Unfolding to expose cells in a user-defined heap structure is necessarily heuristic, but the distinguished traversal parameter in our inductive predicates (i.e., α in $\alpha \cdot \mathsf{c}(\ldots)$) provide guidance (also see our prior work [5] for ways to make unfolding more robust). More crisply, we describe the abstract interpretation of an update with the following inference rule:

$$\boxed{\langle \mathcal{A}, \mathit{statement} \rangle \Downarrow \mathcal{A}'}$$

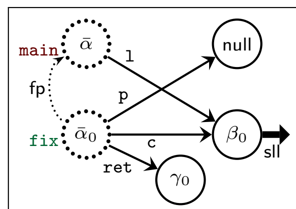
$$\frac{\langle \mathcal{A}, e_1 \rangle \Downarrow \langle \mathcal{A}', \alpha_1 \cdot f_1 \mapsto \beta_1 \rangle \quad \langle \mathcal{A}', e_2 \rangle \Downarrow \langle \mathcal{A}'', \alpha_2 \cdot f_2 \mapsto \beta_2 \rangle}{\langle \mathcal{A}, e_1 := e_2 \rangle \Downarrow \mathcal{A}''[G \to G(\mathcal{A}'')[\alpha_1 \cdot f_1 \mapsto \beta_2]]}$$

The abstract interpretation judgment $\langle \mathcal{A}, \mathit{statement} \rangle \Downarrow \mathcal{A}'$ says that in abstract state $\mathcal{A}$, evaluating statement $\mathit{statement}$ produces a resulting state $\mathcal{A}'$ (as in a standard structured concrete operational semantics). This judgment is defined in terms of an auxiliary judgment $\langle \mathcal{A}, e \rangle \Downarrow \langle \mathcal{A}', \alpha \cdot f \mapsto \beta \rangle$ that evaluates an addressing expression e to a points-to edge in $\mathcal{A}$, which may yield a modified state $\mathcal{A}'$ as the result of unfolding. To reflect the update, we write $G(\mathcal{A})$ for looking up the graph component of $\mathcal{A}$, $A[G \to G']$ for replacing the graph component of $\mathcal{A}$ with G', and $G[\alpha \cdot f \mapsto \beta]$ for updating the edge with source $\alpha \cdot f$ in G to point to β.

3.3 Towards Analyzing Calls and Returns

Our whole-program analysis is based on an abstract interpretation [7] of the program's interprocedural control-flow graph. We desire a sound analysis, which means at each step, the analysis applies locally-sound transfer functions. They cannot omit any possible concrete behavior. With just the abstraction described in this section, we can define an analysis for non-recursive programs (albeit a potentially computationally expensive one).

At a function call site in the concrete execution, a new activation record is pushed onto the call stack. Correspondingly at the abstract level, we push a new abstract activation record: (1) we create a new node representing the base address of the new activation record; and (2) we set the content of its fields (i.e., the formal parameters and local variables) by assigning formal parameters to the actual arguments and by pointing local variables to fresh nodes. For example, consider again the code from Figure 1(a). At the call to `fix` from `main` (i.e., `fix(l, NULL)`) during the analysis, we push a new abstract activation record with base address $\bar{\alpha}_0$ with fields for parameters `c` and `p` and local variable `ret` as shown in the inset. The $\bar{\alpha}_0 \cdot \mathsf{fp}$ field is set to point to $\bar{\alpha}$, the base address of the activation record for `main`, while $\bar{\alpha}_0 \cdot \mathtt{c}$ is made to point to β_0 (as $\bar{\alpha} \cdot \mathtt{l} \mapsto \beta_0$) and $\bar{\alpha}_0 \cdot \mathtt{p}$ gets null. For the $\bar{\alpha}_0 \cdot \mathtt{ret}$ field, it is set to fresh node γ_0, which indicates it contains an arbitrary value. The $\beta_0 \cdot \mathsf{sll}()$ inductive edge states that β_0 is the head of a singly-linked list (of `dll` nodes), which existed before the call.



Now, if we continue analyzing the body of `fix`, we see that $\beta_0 \cdot \mathsf{sll}()$ would be unfolded along the path where $\beta_0 \neq \mathsf{null}$ (i.e., `c != NULL`) before arriving at a recursive call to `fix`. It is clear that if we continue analyzing in the manner described above, we will keep on creating new activation records and never terminate. Thus, in order to ensure the termination of the analysis in the presence of recursive functions, we must apply a *widening* that is capable of summarizing the call stack.

As alluded to earlier, the key observation is that the call stack is itself an inductive structure. Specifically, it is a list of activation records where the frame pointer fp fields form the backbone. Based on this observation, we use a special inductive predicate stack to summarize recursive segments of the call stack (e.g., the sequence of `fix` calls—`fix`*::`main`). This inductive predicate stack is necessarily more complex than usual predicates for summarizing recursive heap structures and is described in detail in Section 4. Furthermore, the stack predicate must be program-specific because it depends on the program's interprocedural control-flow. Thus, its definition cannot be known before beginning the analysis. In Section 5, we detail an algorithm for deriving a definition of stack on the fly during the analysis.

Finally, at a function return site, the analysis proceeds like in a concrete execution by popping off the most recent activation. For instance, in the example above, on return from `fix` to `main`, we drop the node $\bar{\alpha}_0$ and its outgoing edges for fp, c, p, and `ret`, just like the disposal of heap cells. As an invariant of the analysis, we make sure the topmost activation record is always exposed (i.e., never summarized in a stack predicate). This invariant ensures that all program variables in scope (globals and locals) are directly accessible. As part of the transfer function for return, we need to make sure that the activation record of the caller function becomes exposed after the return, as now it is the topmost one. In the presence of call stack summarization that includes the caller, we need to unfold the stack predicate to expose the caller's activation record—using the $\rightsquigarrow_{\textbf{unfold}}$ operation with stack. As such, stack segments always go from callees to callers, that is, in the direction of the fp links.

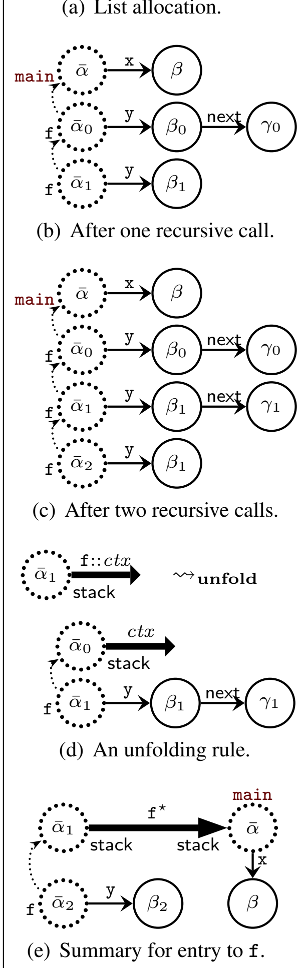
4. Summarizing Call Stacks Inductively

As alluded to earlier, we summarize call stacks from recursive programs using an inductive predicate stack, exploiting their inherent inductive structure. The definition of stack is particularly interesting because it depends on the interprocedural control flow of the program being analyzed. To build intuition for a definition of stack, we first explore a number of examples that illustrate requirements for it. We consider analyzing simple recursive functions requiring no relations between successive activations, recursive procedures requiring simultaneous summarization of the stack and heap, nested recursion, and mutual recursion. Our algorithm for automatically deriving a stack definition on the fly during analysis is then described in Section 5.

Recursion without Heap Relations between Activations. Consider the simple program shown in inset (a) that constructs a list of random length using a recursive function `f`. For each call to function `f`, a new, uninitialized **list** node is allocated on the heap before the next recursive call. For simplicity in presentation, a **list** node has just one field: `next` for linking. Note that we elide the size-of argument to `malloc`, treating it as a high-level allocator with types. The `next` pointers are assigned during the sequence of returns. Inset (b) shows a graph (with no summarization) that describes the state of the program at the entry of the first recursive call to `f` (i.e., with the call string `f::f::main`), while inset (c) shows the state at the second recursive call (i.e., `f::f::f::main`). The repeated pattern is clear from these diagrams: all pending activations of `f` have a local variable `y` that point to a **list** node whose `next` field is arbitrary. This apparent pattern suggests the unfolding rule or case for the definition of `stack` shown in inset (d).

```
void main() { list* x = f(); }
list* f() {
  list* y;
  if (...) return NULL;
  else {
    y = (list*)malloc();
    y->next = f();
    return y;
  }
}
```

(a) List allocation.



(b) After one recursive call.

(c) After two recursive calls.

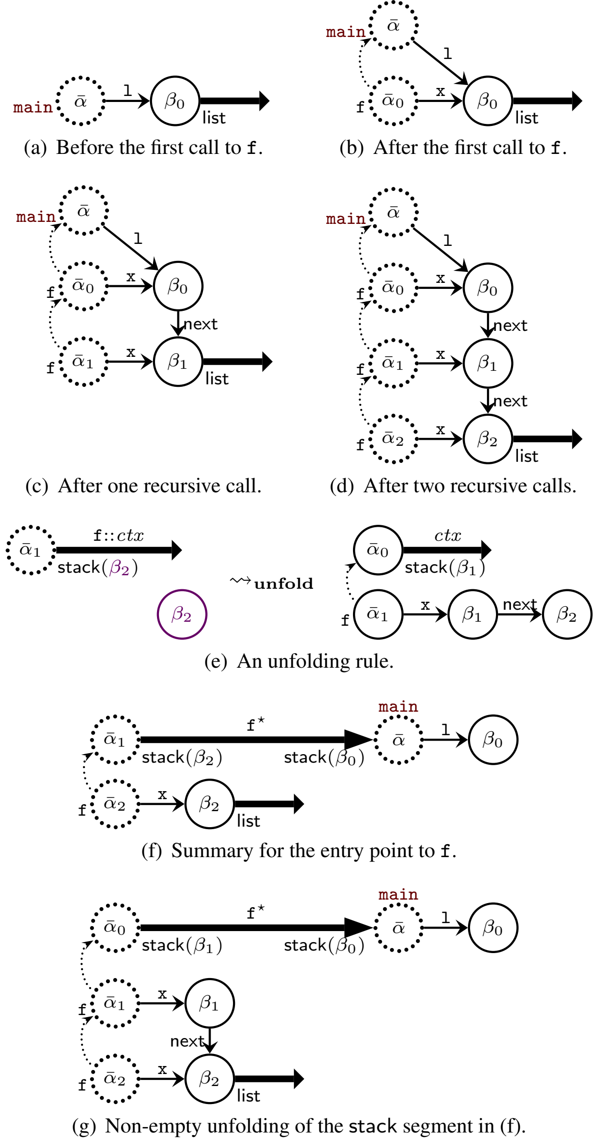
(d) An unfolding rule.

(e) Summary for entry to `f`.

Inductive edges of `stack` are labeled with a regular expression to convey the set of call strings to which the rule can be applied. Here, the regular expression $\mathtt{f}::ctx$ on this rule indicates that it can be applied only to a call stack where the topmost activation record corresponds to an `f` record, while the calling context ctx may be arbitrary. These constraints on the calling context can be expressed as an additional parameter to `stack`, so the labels are simply a shorthand. In other words, the first additional parameter to `stack` is a set of possible call strings expressed as a regular expression. In the case of rule definition, the label is a check for that regular expression pattern (e.g., $\mathtt{f}::ctx$ on the left-hand side of the example $\rightsquigarrow_{\textbf{unfold}}$). For an instance of the `stack` predicate, it gives an abstraction of the call string. As an example, using this rule, we can over-approximate all the possible states at the entry to function `f` after any number of recursive calls with the graph shown in inset (e). The $\mathtt{f}^\star$ label indicates that there are zero or more `f` activations between $\bar{\alpha}_0$ and $\bar{\alpha}$. Intuitively, these labels approximate the sequence of frame pointer links and the types of activation records along that sequence.

Recursion with Mixed Call Stack and Heap Summaries. In Figure 4, we present a slightly more involved example where an existing heap data structure is traversed recursively. Function `f` a recursive, non-destructive walk of a list (i.e., a singly-linked list consisting of **list** nodes). Before the first call to `f`, the memory state is abstracted by the graph shown in Figure 4(a). After the first call

```
void main() { list* l; ... /* make l · list() */ ...; f(l); }
void f(list* x) { if (x == NULL) return; else f(x->next); }
```



(a) Before the first call to `f`.

(b) After the first call to `f`.

(c) After one recursive call.

(d) After two recursive calls.

(e) An unfolding rule.

(f) Summary for the entry point to `f`.

(g) Non-empty unfolding of the `stack` segment in (f).

Figure 4. Summarizing the memory states for recursive list traversal.

to `f` but before any recursive call, we have the graph shown in (b). In (c), we show the graph after the first recursive call at the entry point to `f`, while (d) shows the graph at the same point after two recursive calls. Just like in the previous example, there is a clear repeating pattern consisting of both stack and heap edges for each activation. However, unlike the previous example, there is an important relation between successive activation records through the heap: `x->next` of an activation record aliases `x` of the subsequent activation (e.g., $\bar{\alpha}_0 \cdot \mathtt{x} \mapsto \beta_0 \ast \beta_0 \cdot \mathsf{next} \mapsto \beta_1 \ast \bar{\alpha}_1 \cdot \mathtt{x} \mapsto \beta_1$). Such a relation can be captured with an additional parameter to the `stack` predicate, and thus we get the unfolding rule shown in Figure 4(e). The key difference between the unfolding rule here and the rule from the previous example is the parameter β_2 (shown highlighted in the figure) that says the `next` field from the value of $\bar{\alpha}_1 \cdot \mathtt{x}$ (i.e., β_1) points to an existing node given by parameter

β_2. Using this inductive definition, we can summarize the possible states at the entry to function `f` using a stack segment edge (shown in Figure 4(f).

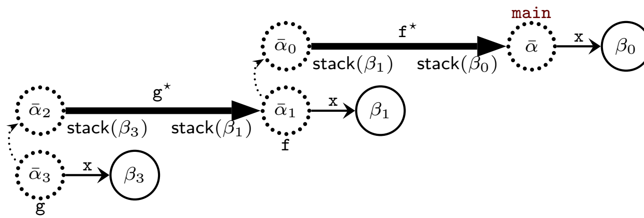
In the graph shown in Figure 4(f), the stack segment not only summarizes a portion of the call stack but also a fragment of the heap, specifically the list segment between β_0 and β_2. It also maintains the relation between the `x` pointers to elements in this list—the inductive definition of stack is quite powerful here. To get a better sense of this aspect, consider unfolding the stack segment in the summary shown in Figure 4(f). There are two cases:

- The segment is empty. This base case says there are no cells summarized by the segment and the ends are equal, that is, $\bar{\alpha}_1 = \bar{\alpha}$ and $\beta_2 = \beta_0$. This state is exactly the one at the entry point after the first call to `f` (i.e., with call string `f`::`main`) shown in Figure 4(b).
- The segment is non-empty. One step of unfolding yields the graph shown in Figure 4(g). Notice that one step of unfolding exposes the previous activation record with the desired relation between the previous activation's `x` and current activation's `x`. Overall, this inductive case summarizes the state after successive recursive calls (e.g., states shown in Figures 4(c) and (d)). For instance, replacing the stack segment in (g) with the empty segment (i.e., unfold it to empty), we get the state after one recursive call (c) where $\bar{\alpha}_0 = \bar{\alpha}$ and $\beta_1 = \beta_0$.

In both examples thus far in this section, we never defined a base case for the stack inductive definition. At the same time, it is not particularly meaningful to provide one, as the first function called in the program is `main`, which must be the first/oldest activation record. For our analysis, this absence of a base case for stack is actually never a problem, as the stack predicate is always used as a segment edge. For any segment, the base case is the empty segment.

Nested Recursion. The program below illustrates a nested recursion: `main` calls function `f`, which calls itself recursively a certain number of times, until it calls `g`, which then also calls itself recursively a certain number of times. There is no call to `f` from `g`.

```
void main() { t* x; f(x); }
void f(t* x) { if (...) f(x); else g(x); }
void g(t* x) { if (...) g(x); }
```

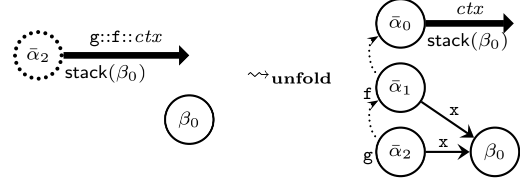


After a number of recursive calls, the layout of the call stack at the entry point to function `g` can be summarized by the above graph where the stack segments correspond to the sequence of recursive calls in `g` and to the sequence of recursive calls in `f`, respectively.

Mutual Recursion. In the program below, functions `f` and `g` are mutually recursive, so the call strings at the entry point to `g` are of the form g::`f`::(g::`f`)*::`main`.

```
void main() { t* x; f(x); }
void f(t* x) { if (...) g(x); }
void g(t* x) { if (...) f(x); }
```

As the cycles are of the form g::`f`::$\cdots$, the call stack of the above program at the entry point to `g` is summarized using the following rule:



The cycles can be more complex. For example, for call stacks where the call strings are of the form g::(`f`*::g)*::`main`, the inductive stack rule for the (`f`*::g)* cycle would unfold into a call stack fragment, which would contain a summary for the inner `f`* cycle (i.e., another stack segment over the `f`* call string).

***Defining* stack.** We can now state precisely the notion of an inductive definition suitable for abstracting the call stack:

- A *stack segment* is a segment edge. This edge is labeled with with a regular expression denoting a superset of the call strings that it describes.
- A *stack inductive definition* is an inductive definition stack such that each case unfolds a sequence of one or more activation records according to a call string.

As stack segments are simply segment edges of the stack definition, the concretization of graphs with stack segments follows from the definitions in Section 3. Notably, given the ability to parametrize inductive definitions by simple regular expressions, the meaning of stack summaries falls directly from the notion of inductive segments.

5. Inferring Call Stack Summarization Rules

In Section 4, we illustrated how the call stack corresponding to various forms of recursion is summarized by an inductive stack predicate. However, we also need to be able to derive a suitable definition for stack.

To obtain a terminating analysis, we require a widening operator capable of summarizing the call stack. To do so, it must fold fragments into stack segments. Yet, before beginning the analysis, the definition of stack cannot be known, as the interprocedural control flow of the program is still to be explored. This circularity means that the definition of stack must be derived on the fly during the analysis when recursive calls are found. In this section, we describe such an algorithm for defining a program-specific stack predicate on the fly.

Widening in Recursive Cycles. In this section, we consider a recursive function `f`, which directly calls itself. The technique we propose also applies to more complex cycles (e.g., mutual recursion). In general, when a function call is recursive, the interprocedural control-flow graph contains two cycles: one at the function entry (from the recursive call site) and one at the function exit (to the recursive return site). Thus, to ensure termination of the analysis in the presence of recursion, widening is applied at the entry and exit points of a recursive function.

We first describe, at a high-level, the steps that the analysis takes to compute an invariant at such a recursive widening point. The key operations are the widening on program states given some stack definition (see Section 6.1) and a *shape subtraction* to generate inductive rules of the stack predicate described later in this section.

Intuitively, deriving rules for stack comes from finding the difference between successive abstract program states at, for example, `f`'s entry point after some number of recursive calls. Specifically, the analysis takes the following steps at `f`'s entry point:

1. Compute a few abstract states by iterating over the recursive call cycle. In practice, unrolling a few iterations of a cycle is

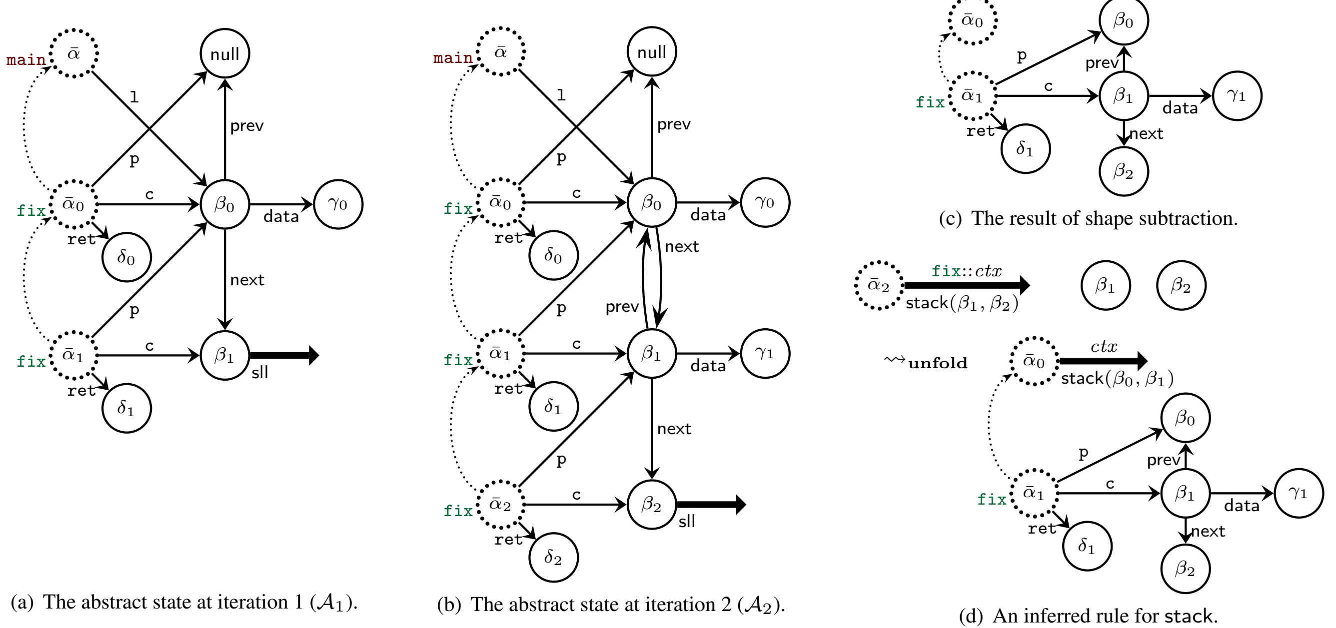


Figure 5. Inferring a stack rule from abstract states computed at the entry point to fix while analyzing the example in Figure 1(a).

often beneficial to get to stable behavior. Suppose we obtain three states $\mathcal{A}_0$, $\mathcal{A}_1$, and $\mathcal{A}_2$ from the first iterations, and we wish to extrapolate from $\mathcal{A}_1$ and $\mathcal{A}_2$.

2. We derive an inductive rule for stack from $\mathcal{A}_1$ and $\mathcal{A}_2$ using the shape subtraction algorithm.
3. We weaken $\mathcal{A}_2$ into a weaker $\mathcal{A}'_2$ using the rule derived in step 2. To perform this weakening step, we apply the widening operator over program states to $\mathcal{A}_1$ and $\mathcal{A}_2$ to produce $\mathcal{A}'_2$. The net result is that $\mathcal{A}'_2$ summarizes both $\mathcal{A}_2$ and $\mathcal{A}_1$ with a stack segment for the difference between them.
4. We perform widening iteration over the cycle until convergence to an invariant $\mathcal{A}_\infty$. The definition of the widening operator over program states guarantees convergence.

Note that the algorithm for inferring new inductive rules for stack in step 2 assists widening in obtaining quality invariants. It does not need to be sound or complete. In fact, it is possible to craft complicated examples where discovering a repeating pattern would be arbitrarily difficult. Instead, the goal should be that it is effective at discovering adequate rules in realistic situations.

Creating* stack *Rules by Shape Subtraction. At a high-level, we find the difference in the graphs between two successive state iterances $\mathcal{A}_1 = (G_1, N_1)$ and $\mathcal{A}_2 = (G_2, N_2)$ using shape subtraction. This difference gives the exposed or unfolded fragment in a new stack rule. In the following, suppose $\mathcal{A}_1$ corresponds to call string $\texttt{f}::\texttt{f}::ctx$ and $\mathcal{A}_2$ to $\texttt{f}::\texttt{f}::\texttt{f}::ctx$.

The first step is to derive the graph part of the new rule. To do so, we want to isolate the edges that appear in G_2 but not in G_1. In other words, we want to partition G_2 into two disjoints sets of edges G_{common} and $G_\triangle$ such that informally speaking,

$$G_2 = G_{\text{common}} * G_\triangle \qquad \text{and} \qquad G_1 = G_{\text{common}} \ .$$

The above is informal because we must take care of matching symbolic node names (which correspond logically to existential variables). We illustrate the description of the algorithm by following the example from Section 2 (i.e., Figure 1). Figure 5(a) shows the abstract state at the entry point to fix obtained from one iteration over the recursive call cycle (i.e., after executing one recursive call); Figure 5(b) shows the abstract state after one more recursive call (i.e., after two recursive calls). The graph shown in Figure 5(c) shows the $G_\triangle$ computed from the states in (a) and (b).

In essence, shape subtraction works by performing a simultaneous traversal over G_1 and G_2 to identify matching structure (i.e., G_{common}). A node naming relation $\Psi \subseteq \mathbb{V}^\sharp \times \mathbb{V}^\sharp$ serves to track the correspondence between the nodes in G_1 and those in G_2, as well as to define the frontier of the traversal. To start the subtraction process, the node naming relation Ψ is initialized with root nodes of memory regions that we want to be in G_{common}. In particular, we pair the following for the initial Ψ: (1) nodes representing addresses of global variables, (2) base addresses of activation records in the context ctx (e.g., $(\bar{\alpha}, \bar{\alpha}) \in \Psi$ and $(\bar{\alpha}_0, \bar{\alpha}_0) \in \Psi$ for initializing the example subtraction in Figure 5), (3) the base address of the topmost activation record (e.g., $(\bar{\alpha}_1, \bar{\alpha}_2) \in \Psi$). This initialization states that G_{common} is any portion of memory reachable from the globals, activation records of the context, and the topmost activation. What remains, $G_\triangle$, is the state difference between G_1 and G_2 that we wish to summarize with a stack segment. In the example, the only activation record node that does not appear in Ψ is the one for the second activation in G_2 (i.e., $\bar{\alpha}_1$ in Figure 5(b)). Observe that this node is exactly the base address of the activation that we wish to summarize.

At this point, the algorithm is rather straightforward. We collect together edges of the same kind whose the source nodes are in the node naming relation Ψ. Whenever two edges are matched, the target nodes (and any additional checker parameters) are added to Ψ. The matched edges are discarded from G_1 and G_2 and added to G_{common} (up to node renaming). For example, in Figure 5, the edges corresponding to field l of $\bar{\alpha}$ can be matched and consumed right after initialization. Then, the pair (β_0, β_0) is added to Ψ, and the prev edges from β_0 in both graphs can be consumed next. We iterate this "match and consume" traversal until G_1 is empty in which case G_2 has become $G_\triangle$.

It is possible that G_1 fails to become empty, that is, we are unable to find a common fragment. This subtraction algorithm is much like the graph join algorithm [6] in that the result depends on the traversal order. Matching and consuming certain edge pairs too early may cause the algorithm to fail to produce an empty G_1 when another traversal order would have succeeded. Fortunately, in our experience, a simple breadth-first-style strategy suffices. First, we match fields from the topmost activation record, which are not pointed to by other activation records directly (e.g., from $(\bar{\alpha}_1, \bar{\alpha}_2)$). Then, we do the same for the fields from the context (e.g., from $(\bar{\alpha}, \bar{\alpha})$ and $(\bar{\alpha}_0, \bar{\alpha}_0)$). Nodes directly pointed to by the activation being summarized are considered last.

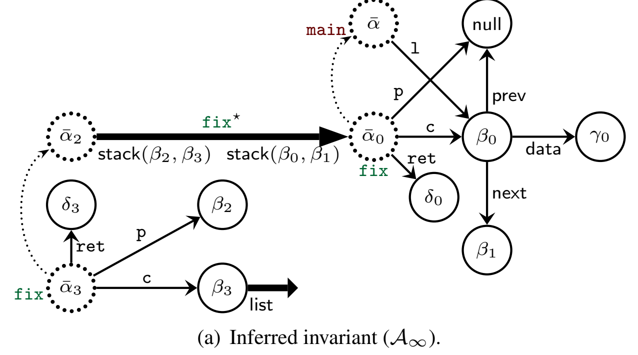
The graph result of shape subtraction G_Δ gives the unfolded edges for a stack unfolding rule, that is, the portion of memory that could be summarized by a stack segment (of length 1). As-is, such a rule does not express all the properties that are needed to describe the call stack precisely. In the Figure 5 example, we need to express aliasing relations between successive activations (cf., the difference between the list allocation and the list traversal examples in Section 4). These relations are captured by parameters on the stack definition. The parameters are given by the nodes at the boundary between the topmost activation and the activations being summarized. In this case, nodes β_1 and β_2 are this boundary and become parameters in the definition of the stack rule as shown in Figure 5(d). Finally, any relevant numerical constraints in the base domain is also captured in defining a stack rule. Once G_Δ has been computed, we simply take the projection of the numerical invariant N_2 onto the set of symbolic node names in G_Δ.

Shape subtraction on the graph portion can be seen as a restriction on frame inference [1]. Here, we are looking for an exact match as opposed to an entailment between two configurations. In spirit, the above algorithm potentially could be applied to derive other kinds of inductive definitions besides stack (cf., [14]). However, we make critical use of understanding the inductive structure of a call stack to get good results. For example, this background knowledge is used in initializing the node naming relation Ψ. We hypothesize that having some knowledge on the kind of inductive backbone of interest is key to getting high-quality definitions.

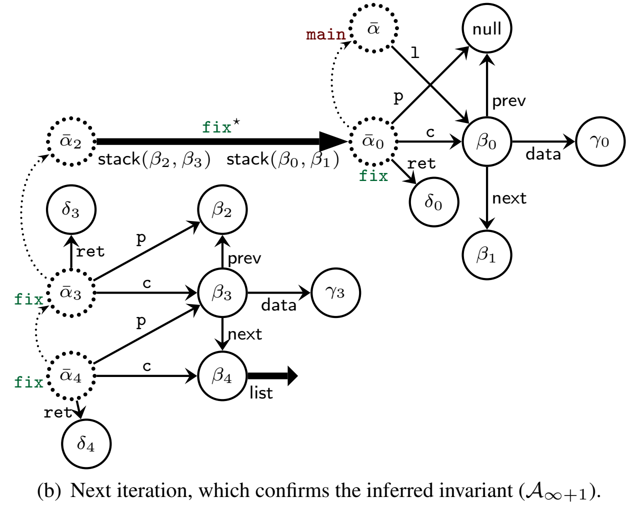
6. Applying Call Stack Summaries in Analysis

With the mechanism for deriving stack rules during analysis, we have all the pieces for analyzing recursive procedures with call stack summarization by following the outline in Section 3.3. In particular, sound transfer functions from intraprocedural inductive shape analysis [5, 6, 18] for basic program statements, like assignment (cf., Section 3.2), guard conditions for branching, loops, and memory allocation-deallocation, carry over in a straightforward manner. A slight difference is that instead of a fixed set of variables as in the intraprocedural case, we have both global and local variables. Local variables are fields of activation records in our graph, but all program variables in scope are easily accessible, as we ensure that the topmost activation is never summarized.

There are two remaining pieces to our interprocedural analysis. First, we want to see how widening with derived stack rules applies at the function entry and exit points in recursive call cycles (Section 6.1). Second, the soundness and termination of extensible inductive shape analysis [5, 6] relies on the assumption that all inductive definitions are fixed before the analysis starts. In this paper where the definition of stack is extended on the fly, we need to justify soundness and termination in the presence of such on-the-fly inductive rule generation (Section 6.2). We conclude this section with a summary of the reasons for termination and soundness for the overall analysis, as well as some empirical experience (Section 6.3).



(a) Inferred invariant ($\mathcal{A}_\infty$).



(b) Next iteration, which confirms the inferred invariant ($\mathcal{A}_{\infty+1}$).

Figure 6. Widening at the entry point to `fix` in Figure 1(a).

6.1 Widening with Call Stack Summaries

After an appropriate inductive rule for stack has been derived (see Section 5), widening at function entry and exit points in a recursive call cycle is not particularly different than at loop heads. That is, the join $\sqcup$ and widen ∇ on graphs and program states [6] essentially carries over. We do not redefine these algorithms, but we discuss their main features here by following our example introduced in Section 2.

The join on separating shape graphs is stabilizing, so the only difference between join $\sqcup$ and widen ∇ is the operator applied to elements of the numerical base domain. At a high-level, the join on graphs is actually quite similar to the subtraction algorithm described in Section 5. They both work by a simultaneous match and consume traversal over two graphs from root nodes using a node naming relation (i.e., Ψ). Roughly speaking, the main difference is that join applies weakening to memory regions delineated by the traversal. For example, it folds fragments consisting of points-to edges (e.g., $\alpha \cdot \mathsf{next} \mapsto \beta$) into an instance of an inductive definition (e.g., $\alpha \cdot \mathsf{list}()$). Folding is in essence applying an unfolding rule in reverse [6].

Following the outline in Section 5, at the entry point to `fix`, we first obtain abstract states $\mathcal{A}_1$ and $\mathcal{A}_2$ from Figures 5(a) and (b), respectively. Subtraction is applied to them to get the stack rule in Figure 5(d). With this rule, widening on abstract states is applied to $\mathcal{A}_1$ and $\mathcal{A}_2$ to produce the state $\mathcal{A}_\infty$ shown in Figure 6(a), which summarizes both $\mathcal{A}_1$ and $\mathcal{A}_2$. Beginning at the entry point to `fix` with $\mathcal{A}_\infty$, we analyze until the next recursive call and returning to the entry point, we get the abstract state $\mathcal{A}_{\infty+1}$ shown

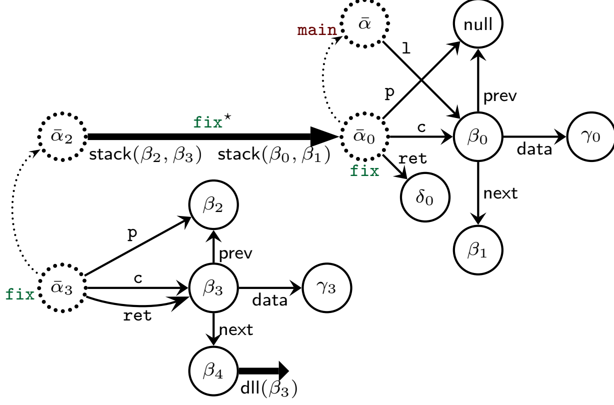


Figure 7. An invariant just before returning from a recursive call to `fix`.

in Figure 6(b). Observe that $\bar{\alpha}_3$ is the activation record that can be folded into the stack segment (using the Figure 5(d) rule). Thus, computing $\mathcal{A}_\infty \nabla \mathcal{A}_{\infty+1}$ yields $\mathcal{A}_\infty$, which confirms that $\mathcal{A}_\infty$ is an invariant that summarizes the set of all concrete states that can be observed at the entry to function `fix` (after one or more recursive calls).

The control point after the function exit before the return site of a recursive call is also on a cycle in the interprocedural control flow graph, so we perform widening iterations there. We perform widening at the point just before the topmost abstract activation record is discarded. In Figure 7, we show an abstract element summarizing concrete states that can be observed just before returning from a recursive call to `fix` (and where this call went along the path where c was not null and node c was not removed). Note that during the sequence of returns from recursive calls, the new dll edge appears, as upon function return, the tail of the structure has prev pointers set correctly so as to define a dll.

6.2 Introducing Inductive Rules on the Fly

As noted above, a source of complexity in summarizing the call stack with the stack predicate is that new inductive rules are generated on the fly. To reason about this aspect, we extend our framework with rule set extension. When the analysis starts, the set $\mathcal{R}$ is empty. Whenever a new recursive call site is discovered, a new rule is added to $\mathcal{R}$. Therefore, we consider $\mathcal{R}$ an element of a separate lattice, specifically the powerset of the set of rules. Each invariant is with respect to a set of rules $\mathcal{R}$, which determines the instance of the graph domain in use. Thus, our analysis state is actually an $(\mathcal{R}, G, N)$ tuple in the abstract domain $\mathbb{D}^\sharp$ defined below. For clarity, we annotate $\mathbb{D}^\sharp_{\mathrm{graph}}$ with the set of rules $\mathcal{R}$ allowed for the inductive definition of stack as follows: $\mathbb{D}^\sharp_{\mathrm{graph}\langle\mathcal{R}\rangle}$ (and similarly with $\boldsymbol{\gamma}_{\mathrm{graph}\langle\mathcal{R}\rangle}$). We also define $\boldsymbol{\gamma}_{\langle\mathcal{R}\rangle}(G, N)$ as we did $\boldsymbol{\gamma}(G, N)$, except we now use $\boldsymbol{\gamma}_{\mathrm{graph}\langle\mathcal{R}\rangle}$ in place of $\boldsymbol{\gamma}_{\mathrm{graph}}$ (and similarly for the order $\sqsubseteq_{\langle\mathcal{R}\rangle}$ on states (G, N)).

$$\mathbb{D}^\sharp \stackrel{\mathrm{def}}{=} \{ (\mathcal{R}, G, N) \mid G \in \mathbb{D}^\sharp_{\mathrm{graph}\langle\mathcal{R}\rangle} \text{ and } N \in \mathbb{D}^\sharp_{\mathrm{num}} \}$$

$$\boldsymbol{\gamma}(\mathcal{R}, G, N) \stackrel{\mathrm{def}}{=} \{ s \mid s \in \boldsymbol{\gamma}_{\langle\mathcal{R}\rangle}(G, N) \}$$

$$(\mathcal{R}_0, G_0, N_0) \sqsubseteq (\mathcal{R}_1, G_1, N_1) \quad \text{iff} \quad \mathcal{R}_0 \subseteq \mathcal{R}_1 \text{ and } (G_0, N_0) \sqsubseteq_{\langle\mathcal{R}_1\rangle} (G_1, N_1)$$

The above ordering $\sqsubseteq$ is sound, as $\mathcal{R}_0 \subseteq \mathcal{R}_1$ implies that $\boldsymbol{\gamma}_{\mathrm{graph}\langle\mathcal{R}_0\rangle}(G_0) \subseteq \boldsymbol{\gamma}_{\mathrm{graph}\langle\mathcal{R}_1\rangle}(G_1)$. All transfer functions are as before, except for widening at the head of recursive functions, which also adds a new rule. Most importantly, as shown in Venet [29] that has similar a construction, this widening stabilizes if the process of adding rules is itself bounded. One possible bound is to allow at most one rule per call site.

The above construction also suggests applying more complex forms of widening to the set of rules $\mathcal{R}$, while preserving soundness. In the ordering $(\mathcal{R}_0, G_0, N_0) \sqsubseteq (\mathcal{R}_1, G_1, N_1)$ defined above, we stated that it must the case that $\mathcal{R}_0 \subseteq \mathcal{R}_1$. However, we could use a more sophisticated ordering on sets of rules. In particular, if we have two rules r_0 and r_1 where r_1 is weaker than r_0, then we could replace r_0 with r_1 in our set of rules while maintaining soundness. In terms of the analysis, this observation means that inductive rules for stack may be weakened during the course of the analysis. Surprisingly, we can use this process to improve what can be summarized. Suppose the analysis discovers a stack rule r to summarize the call stack at the entry of some function f. However, widening at the next iteration fails (i.e., is imprecise) due to rule r being too specific, we are allowed to weaken r to a coarser rule r' that may allow this widening step to succeed. This technique is potentially useful when the shape part of the rules is stable, but when the numeric contents of cells need to be computed by a non-trivial widening sequence, as can be seen in Section 7. To guarantee termination of the analysis with this process, the rule weakening step must be shown to stop in some way.

Benchmark	Recursive (ms)	Iterative (ms)
list traversal	11	4
list get n*th* element	22	4
list insertion n*th* element	48	16
list remove n*th* element	27	11
list deletion (memory free)	13	4
list append	20	13
list reverse	29	5

Table 1. Micro-benchmarks comparing analysis times for recursive and iterative versions of the same operation.

6.3 Termination, Soundness, and Empirical Experience

To ensure termination, the analysis algorithm applies widening to at least one point in each cycle in the set of abstract flow equations. In the case of whole-program interprocedural analyses, the following is one set of such widening points: (1) loop heads for intraprocedural loops and (2) at the entry and at the exit of functions when analyzing a recursive call. At the end of the analysis, each program point is mapped to a finite set of abstract elements. As all transfer functions are sound, and there is at least one widening point on each cycle in the interprocedural control-flow graph, the analysis terminates and is sound:

Soundness. *If concrete state s can be reached at program point ℓ and if the set of abstract elements computed for point ℓ is $\{ (\mathcal{R}^\ell_0, G^\ell_0, N^\ell_0), \ldots, (\mathcal{R}^\ell_n, G^\ell_n, N^\ell_n) \}$, then $s \in \boldsymbol{\gamma}(\mathcal{R}^\ell_i, G^\ell_i, N^\ell_i)$ for some $i \in 0..n$.*

Preliminary Empirical Experience. We have implemented shape subtraction and stack rule inference described in Section 5 in Xisa [5, 6]. It discovers the appropriate stack rules for all of the examples given in Sections 2 and 4. In each case, the stack rule inference time is negligible. We also have implemented a prototype analyzer and ran it on a series of micro-benchmarks that compares the analysis time of some recursive functions against their iterative counterparts (Table 1). The tests were performed on a 2.4 GHz MacBook Pro with 8 GB of RAM and under a Linux 2.6.27 virtual machine. In all these cases, the memory usage is not significant (at most 6 MB).

While the analysis times on these micro-benchmarks are negligible, we do see that analyzing the recursive versions take about two to three times more time than their iterative counterparts. This slowdown is expected since the analysis of a recursive function involves not only the inference of a suitable stack definition but also two fixed-point computations (one over the call sites and one over the return sites). In contrast, the analysis of a single imperative loop requires only one fixed point computation. Note that in the case of list reverse, the imperative version is quite trivial (e.g., does not even require a segment summary in the loop invariant), while the stack inductive definition for the recursive version is just as complex as the other examples.

7. Case Study: Precision and Modularity

In this section, we look more closely at numerical properties, which are combined with shape properties in the presence of recursion. Figure 8(a) shows an example program that implements a filter equation over a doubly-linked list. Here, we replaced the integer data field with two floating point fields x and y. It walks through the structure and deletes the nodes where the x field is not in range $[-M, M]$. In the same pass, it integrates a filter equation by reading from the x fields and writing the result to the y fields; this computation is performed only for the nodes that are not deleted.

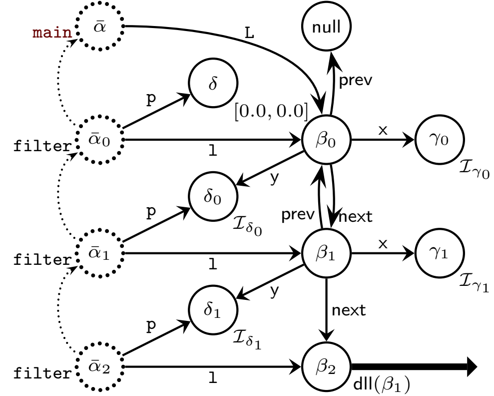
Let us first consider the purely numeric part of the code (i.e., the digital filter) and imagine that it is implemented using a while-loop instead of recursion. On this version of the program, the Astrée analyzer [3] can infer rather precise invariants using either the domain of Feret [11] or simply the interval domain with threshold widening. For instance, if we let $M = 5$, $a = 0.8$, and p is initially set to 0, then Astrée computes the range $[-85.00008, 85.00008]$ as an approximation for the terms of the sequence after 7 iterations using only the interval domain with threshold widening (on the iterative, purely numeric version). Even though Astrée does not support recursion, the basic techniques are applicable to the purely numeric part of the program shown in Figure 8(a).

On the other hand, using a modular approach to interprocedural analysis to reason precisely about the numeric part is quite challenging, as we must capture relations between the inputs and outputs in the abstract domain. For instance, to obtain the same properties as using the interval domain in the iterative version, we must use a complex domain like polyhedra [8]. For non-linear filters like those handled by Feret [11], even polyhedra would not be sufficient to achieve the same level of precision with the modular approach. In essence, relational abstract domains are sometimes required to achieve the same level of precision using the modular approach as non-relational domains on an iterative counterpart.

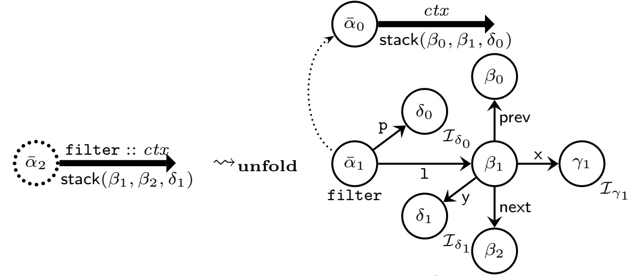
We now consider analyzing the program shown in Figure 8(a) using our call stack abstraction and by instantiating the base domain with intervals, that is, with a simple non-relational domain. When applied to this program, our analysis must infer inductive rules for stack at two call sites (as there are two recursive call sites). We consider only the second call site, as the other one is very similar, though slightly more complicated with respect to shape. Figure 8(b) depicts the abstract state after two recursive calls before summarization of the call stack is performed. Nodes that denote floating point values are annotated with an interval. In particular, we have that $\mathcal{I}_{\gamma_0} = \mathcal{I}_{\gamma_1} = [-M, +M]$, $\mathcal{I}_{\delta_0} = [-a \star M - \epsilon, +a \star M + \epsilon']$, and so on. The ϵ, ϵ' values account for rounding errors. To obtain an invariant, we first derive a stack rule shown in Figure 8(c). This rule allows us to fold the call stack at the first recursive call site. Interestingly, this rule does not work at the next iteration because the interval constraints over the floating point nodes have not yet stabilized. Instead, we weaken the stack as discussed in Section 6.2. In essence, this process amounts to the same series of widening steps on the numerical domain elements as

```
void filter(dll* l, float p) {
  if (l != NULL) {
    if (l->x < -M || l->x > M) {
      dll* n = l->next; remove(l); filter(n, p);
    }
    else {
      l->y = a * p + l->x; filter(l->next, l->y);
    }
  }
}
```

(a) Recursive implementation of a digital filter.



(b) An abstract state after two recursive calls.



(c) An inferred rule for stack.

Figure 8. Obtaining numerical properties on a recursive program with the call stack abstraction.

in the analysis of the iterative program by Astrée, except they are applied when generating the stack summarization rule. Thus, it will lead to the computation of the same numerical invariants.

Recall that we are not advocating an abandonment of modular interprocedural analysis. As noted in Section 1, modularity is often the basis for scalability. When functions should be analyzed out of context (e.g., library API functions), then the modular approach seems ideal. Instead, we present interprocedural analysis by call stack summarization as an alternative that can be more effective in certain situations. A potentially hybrid approach could be to apply modular analysis except were call stack summarization is absolutely required. For example, call stack summarization is applied for a few intricate internal functions as part of a modular, tabulation-based analysis of the program.

8. Discussion: Interprocedural Shape Analysis

In this section, we consider known challenges in the context of interprocedural analysis of heap manipulating programs and comment on trade-offs.

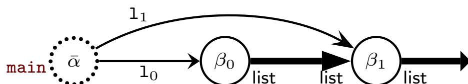
8.1 Cutpoints

Modular shape analyses [4, 12, 13, 16, 23] typically infer an abstraction of the *effect* of a procedure on a portion of the heap. To make this effect abstraction precise, the analysis needs to carefully extract the fragment of the heap that may be modified during a call so that the rest of the heap is a *frame* [21]. Such an analysis must also abstract *cutpoints* [23] carefully to obtain precise results. Cutpoints are the locations at the border of the callee's reachable heap, that is, any node that is reachable from the callee's parameters (though not directly pointed-to by them) and reachable from a pending activation or a global (without following links inside the callee's reachable heap). It is important to track cutpoints precisely in a modular analysis, as they are used to reflect the effect of the callee in the caller's state on function return. Doing so can be challenging, as an unbounded number of cutpoints may arise either due to unbounded recursion or due to the traversal of unbounded heap structures. While there is no general solution for cutpoints today, several partial solutions have been proposed. For example, they focus on isolating their effect by proving cutpoint-freeness [24] or by proving that cutpoints are not live [17], or they reason up to a bounded number of them [23].

Our analysis based on call stack summarization also needs to cope with cutpoints so that the widening iteration reaches a precise fixed point. In the following, we consider how cutpoints impact our analysis in particular situations.

Summarizing Cutpoints. Cutpoints often arise when a field of a caller's activation record point into the heap space that may be modified by the callee (or a subsequent sub-call). This situation arises in the example in Section 2 (Figure 1). Such cutpoints may be unbounded, as each recursive call defines a set of local cutpoints and the number of recursive calls may be unbounded; however, they can be summarized as part of the call stack and re-exposed when recursive calls return. Observe that such cutpoints appear among the parameters in inductive rules for `stack`. Therefore, our approach can deal with an unbounded number of cutpoints by folding them as part of the call stack with finitely many parameters.

Case Splitting from Cutpoints. Another case is when the cutpoints stem from the global variables or from sections of the call stack that are not summarized (*e.g.*, as the activation record of `main`). Such cutpoints cause singularities in the call stack as in the example from Rinetzky et al. [23]. In that example, a singly-linked list is built by appending two singly linked lists while keeping a pointer to the element in the middle; then a destructive reverse function is applied to the whole structure. The state before the list reverse can be summarized as follows:



Now, our analysis needs to walk through the segment between β_0 and β_1 (using the segment unfolding [5]) and discovers a `stack` rule for summarizing the call stack. The generated rule is actually very similar to the one derived from the list traversal example of Section 4 (Figure 4(e)); it is essentially the same except for the direction of `next` pointers, which are switched by the reverse function. As we might predict, special care must be taken for the case where the reverse reaches node β_1. Applying widening here would cause the property that $\mathtt{l}_1$ points inside the list to be lost. A solution is to case split and not perform widening on that iteration. The trick here is a variation on a classical one: when a new behavior is observed (e.g., reaching β_1), postpone widening to the next iteration (cf., [3]).

To summarize, cutpoints induce a somewhat different set of issues compared with modular shape analyses, as our analysis abstracts only states and not effects. These issues can be alleviated using standard techniques to some extent.

8.2 Heap Partitioning and Cutpoints

```
void f(dll* l) {
  if (...) { f(l->prev); }
  else { f(l->next); }
}
```

Sharing can introduce more severe problems, as the example shown inset demonstrates. In this code, function `f` takes as input a doubly-linked list and performs a non-deterministic walk over it: at each step, depending on a random test, it either walks one step forward and calls itself recursively, or it walks one step backwards and calls itself recursively. The difficulty here is that this function may traverse several times the same elements of the doubly-linked list. Thus, at any time in a concrete execution, an element that has been visited in the past may be visited again in the future. This prevents the heap structure from being partitioned into sub-regions touched by successive recursive calls. This problem would also occur in the case of analyses that infer the footprint of procedures so as to compute their effect.

This sort of problem can arise even without procedures, so we consider this issue orthogonal to the cutpoint problem: cutpoints aim at describing the borders in the heap partition used to abstract successive calls, whereas the issue of this example is that no simple and good partitioning exists.

9. Related Work

The most closely related work to ours is Rinetzky and Sagiv [22]. They designed an analysis based on three-valued logic [27] where the call stack is abstracted together with the heap. Some earlier work had similar views (e.g., [9]). Rinetzky and Sagiv [22]'s concrete model is quite similar to ours. However, the abstraction is radically different, as we use separation logic and inductive definitions. In particular, we found that these tools are natural for abstracting the call stack. First, the call stack can be defined inductively, and second, there are built-in separation constraints between the call stack and the heap, as well as between successive activation records. Moreover, we found that the operations like function call, function return, and widening encode well in this abstraction.

Most interprocedural shape analyses take advantage of functions in order to achieve modularity. For instance, Jeannet et al. [16] abstracts input-output relations in the TVLA framework using three-valued structures with two occurrences of each element of the universe: one for the input state and one for the output state. The analysis of Rinetzky et al. [23, 24] is based on the tabulation of pairs of input-output three-valued structures. In Marron et al. [19], cutpoints are used in order to segment the abstract states and restrict the analysis to a precise approximation of the footprint of procedures. Similarly, separation logic-based modular analyses, such as Gotsman et al. [12], Calcagno et al. [4], or Gulavani et al. [13], isolate the footprint of procedures and compute the effect on the local heap (though in different manners). As we noted in Section 1, modularity in shape analysis has many advantages, especially when analyzing libraries or for scalability reasons. Our approach fills a gap to tackle a family of analysis problems, such as the combined analysis example of Section 7, where whole program analysis appears more suitable.

Another area of static analysis related to our work is that of context-sensitive analyses. The term "context sensitive" is used in many ways and may cover very different levels of abstraction, ranging from approaches that use call-strings to describe contexts [28] or an abstraction of the control flow history [25] to techniques where the call stack content is abstracted [15, 22]. Our technique uses a rather coarse abstraction of the call string and of the control flow history, so its precision can be further improved by applying

those techniques [25, 28]. The analysis of Jeannet and Serwe [15] features a non-relational abstraction of the call stack in the sense that it discards the order of the activation records.

10. Conclusion

In this paper, we have explored a whole-program approach to interprocedural analysis over recursive programs. In particular, we have shown how to apply a shape analysis abstraction based on separation logic and inductive definitions to directly summarize call stacks along with heap structures. To automatically derive rules for call stack summarization, we exploited its built-in inductive structure. The XISA framework [5] turned out to be both expressive and robust in abstracting a very concrete model of calling contexts.

Acknowledgments

We would like to thank Josh Berdine, Noam Rinetzky, and Hongseok Yang for interesting discussions on interprocedural shape analysis and on the issues related to cutpoints. We also thank the anonymous reviewers for their helpful suggestions, including insightful comments on the whole-program versus modular approach.

References

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 52–68, 2005.

[2] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Computer-Aided Verification (CAV)*, pages 178–192, 2007.

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

[4] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages (POPL)*, pages 289–300, 2009.

[5] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Principles of Programming Languages (POPL)*, pages 247–260, 2008.

[6] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Static Analysis (SAS)*, pages 384–401, 2007.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–97, 1978.

[9] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Principles of Programming Languages (POPL)*, pages 157–168, 1990.

[10] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302, 2006.

[11] Jérôme Feret. Static analysis of digital filters. In *European Symposium on Programming (ESOP)*, pages 33–48, 2004.

[12] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis (SAS)*, pages 240–260, 2006.

[13] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom-up shape analysis. In *Static Analysis (SAS)*, pages 188–204, 2009.

[14] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *Programming Language Design and Implementation (PLDI)*, pages 256–265, 2007.

[15] Bertrand Jeannet and Wendelin Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Algebraic Methodology and Software Technology (AMAST)*, pages 258–273, 2004.

[16] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2), 2010.

[17] Jörg Kreiker, Thomas Reps, Noam Rinetzky, Mooly Sagiv, Reinhard Wilhelm, and Eran Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. Technical report, Queen Mary University of London, 2010.

[18] Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. Separating shape graphs. In *European Symposium on Programming (ESOP)*, pages 387–406, 2010.

[19] Mark Marron, Manuel V. Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *Compiler Construction (CC)*, pages 245–259, 2008.

[20] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[21] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[22] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. In *Compiler Construction (CC)*, pages 133–149, 2001.

[23] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Principles of Programming Languages (POPL)*, pages 296–309, 2005.

[24] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Static Analysis (SAS)*, pages 284–302, 2005.

[25] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[26] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

[27] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3): 217–298, 2002.

[28] 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, 1981.

[29] Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Static Analysis (SAS)*, pages 366–382, 1996.