Tel-Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
School of Computer Science

INTERPROCEDURAL AND MODULAR LOCAL HEAP SHAPE ANALYSIS

by

Noam Rinetzky

under the supervision of
Prof. Mooly Sagiv

Thesis submitted
for the degree of Doctor of Philosophy

Submitted to the Senate of Tel-Aviv University
June 2008

To my loving and beloved parents, Shalva and Michael.

"Art consists of limitation.
The most beautiful part of every picture is the frame."

–G. K. Chesterton

Abstract

Interprocedural and Modular
Local Heap Shape Analysis

Noam Rinetzky
Doctor of Philosophy
School of Computer Science
Tel-Aviv University

Shape analysis algorithms statically analyze a program to determine information about the heap-allocated data structures that the program manipulates. Traditionally, shape analysis algorithms abstract *whole* heaps at every program point. This makes it difficult to benefit from procedural abstraction in the analysis and to support modular reasoning.

In this thesis, we develop interprocedural and modular *local-heap* shape analysis algorithms for sequential imperative programs. More specifically, we present (i) interprocedural shape analysis algorithms that treat procedures as heap-transformers between *relevant parts* of the heap, and (ii) modular shape analysis algorithms that separately analyze every program module while considering only the subheaps manipulated by that module.

The distinguishing aspect of our interprocedural analyses is that they maintain information regarding the sharing patterns between the procedure local-heap and the other (irrelevant) parts of the heap. Specifically, our analyses take special care of *cutpoints*, objects that separate the "local-heap" that can be mutated by a procedure from the rest of the heap, which—from the viewpoint of that procedure—is non-accessible and immutable. This procedure-local view of the heap allows our analyses to benefit from procedural abstraction.

We also introduce a new modular shape analysis. The distinguishing aspect of our modular analyses is that they integrate a shape analysis with encapsulation constraints. More specifically, we focus our attention on analyzing *dynamically encapsulated* programs. In these programs, the live references (i.e., used before set) between subheaps manipulated by different modules form a tree. Our work presents a nice interplay between dynamic encapsulation and modular shape analysis: it uses dynamic encapsulation to enable modular shape analysis, and uses shape analysis to determine that the program is dynamically encapsulated. Thanks to this interplay, we are able to provide the first *modular* shape analysis algorithm capable of handling pointer parameters.

Technically, we develop our analyses by abstract interpretation of non-standard local-heap concrete operational semantics. These semantics take special care of cutpoints and check various restrictions on programs. Our analyses are simplified by *assuming* these restrictions, yet remain sound by *verifying* them. Furthermore, in addition to designing new cutpoint-aware shape abstractions, we show how placing certain restrictions on the allowed cutpoints *enables* lifting some existing abstractions which have been used for *intra*procedural shape analyses to the *inter*procedural and *modular* setting.

We have implemented our interprocedural algorithms and applied them to verify interesting properties of heap-intensive programs, including the first fully-automatic proof of the partial correctness of a recursive quicksort implementation.

Acknowledgements

I had a great privilege of being a student of Mooly Sagiv and a member of the wonderful group that he formed in Tel Aviv University. Mooly, with endless care and patience, has been a real beacon of optimism, guiding me through the (sometimes rough) seas of research. I would like to thank all the members of Mooly's group for their help and support throughout the years and, most of all, for their friendship. I would like to give special thanks to Eran Yahav for long years of cooperation, fruitful discussions, and moral support. I enjoyed co-authoring the different papers [RBR+05, RSY05a, RPHR+07] on which this thesis is based with Ramalingam, Joerg Bauer, Arnd Poetzsch-Heffter, Tom Reps, and Reinhard Wilhelm. I am also grateful for the anonymous referees of these papers. I am grateful for the helpful comments of Daphna Amit, Nurit Dor, Steven Fink, Tal Lev-Ami, Aant Lotan, Roman Manevich, Shriram Rajamani, Ramalingam, Jan Reineke, Ran Shaham, Greta Yorsh, Eran Yahav and Gilit Zohar-Oren who read earlier drafts of this thesis and the papers from which it is drawn. I would like to thank the anonymous referees of this thesis for their helpful comments.

Noam Rinetzky was supported in part by the German-Israeli Foundation for Scientific Research and Development (G.I.F.), in part by the IBM Ph.D. Fellowship Program, and in part by a grant from the Israeli Academy of Science.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern programs rely significantly on the use of dynamically-allocated linked data structures. Shape-analysis algorithms [Rey68, JM81, JM82] statically analyze a program to determine information about the heap-allocated data structures that the program manipulates. The algorithms are *conservative* (sound), i.e., the discovered information is true for every input. Thus, they can be utilized for program verification, optimization, etc..

Designing shape analysis algorithms, in particular ones which are precise enough for the purpose of *program verification*, is a challenging problem: (i) there is no a priori bound on the number of dynamically allocated objects; and (ii) updates to the state of the program may have indirect effects due to pointer aliasing. Furthermore, handling the heap in a precise manner requires *strong pointer updates* [CWZ90] which calls for the use of flow-sensitive context-sensitive analysis and expensive heap abstractions. The presence of procedures escalates the problem because the analysis needs to track interactions between the program (unbounded) stack and the (unbounded) heap [RS01]. Still, reasoning about the effects of procedure calls is a crucial element in program analysis.

In this thesis, we present shape analysis algorithms that are precise enough for the purpose of *verifying* imperative programs that manipulate *recursive* dynamically-allocated linked data structures using (possibly recursive) *procedures*. Our analyses verify properties such as *memory safety*, e.g., that a program never dereferences a null-valued pointer; establish shape invariants, e.g., that at a certain program point variables x and y always point to disjoint acyclic linked lists; and prove that the analyzed program satisfies certain partial correctness specification, e.g., that a procedure invoked on an acyclic linked list always returns a sorted acyclic linked list.

Our algorithms compute a characterization of a procedure's behavior in which parts of the heap not relevant to the procedure are ignored. We provide two kinds of algorithms:

- *interprocedural* algorithms that analyze whole program and determine properties of *specific programs*, and
- *modular* algorithms that separately analyze *parts* of programs (modules), and identify properties of the data structures manipulated by the analyzed module that hold in any program (which satisfies certain, modularly checkable, conditions).

Technically, we develop our analyses following the semantics-based *abstract-interpretation* [CC77, CC79] approach for program analysis: We define program semantics at multiple levels of abstraction, starting from a concrete (standard) program semantics and ending with the abstract semantics used by the analyzer, and formally establishing the relationship between the different semantics. This allows us to formally derive a sound analysis. In our semantics, procedures are only passed *parts* of the heap. By being abstract interpretations of these semantics, our analyses are able to compute characterizations of procedures as *local-heap* transformers.

By design, instead of attempting to develop algorithms that aim at verifying any (shape) properties of any program, we focus on by designing static shape analysis algorithms aimed at verifying chosen classes of properties for selected classes of programs. Thus, we develop our analysis using a two-staged approach:

I. In the first stage, we identify an interesting class of programs and an interesting class of properties. We define a *non-standard* concrete operational semantics which characterizes the class of programs that we are interested in using semantically-defined conditions. Our non standard semantics agree with the standard semantics with respect to the properties of interest as long as the program belongs to the targeted class of programs. However, our semantics halt the execution of a program if the execution violates the intended restrictions.

II. In the second phase, we develop shape analysis algorithms by abstract interpretation of our non standard semantics. Our algorithms *assume and (conservatively) verify* that the analyzed program belongs to the targeted class of programs. (In particular, by not requiring an a priori classification that a program belongs to the targeted class of program, our analyses can be applied to arbitrary programs). Instead of designing new shape abstractions, we show how to lift existing (bounded) shape abstractions which have been used for *intra*procedural shape analyses (e.g, [DRS00, LARSW00, SRW02, MYRS05, DOY06, LAIS06]) to the interprocedural and modular setting.

1.1 Main Contributions

The main contributions of this thesis can be summarized as follows:

Procedure local-heap Storeless Semantics. We present two procedure local-heap storeless semantics: $\mathcal{LSL}^{\text{CPF}}$ (see Chapter 3) and $\mathcal{LSL}$ (see Chapter 4). In both semantics, called procedures are only passed *parts* of the heap. The two semantics differ in the way that they treat *cutpoints*, objects that separate the "local-heap" that can be mutated by a procedure from the rest of the heap, which—from the viewpoint of that procedure—is non-accessible and immutable. (See Definition 3.3.2).

- $\mathcal{LSL}^{\text{CPF}}$ forbids cutpoints. Specifically, it aborts when a procedure invocation yields a cutpoint.
- $\mathcal{LSL}$ allows arbitrary cutpoints by treating them differently than other, non-cutpoint, objects. More specifically, $\mathcal{LSL}$ marks each cutpoint with an immutable *cutpoint-label*. The cutpoint-label is computed based on of the position of the cutpoint in the procedure's local-heap when the procedure starts executing. As a result, each cutpoint can be identified by a unique canonic context-independent label. (See Definition 4.3.1)

Both semantics preserve procedure-*local* views of the memory. In particular, they preserve the values computed by arbitrary code blocks and program expressions. Thus, abstract interpretation algorithms which are based on $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ can verify program assertions with respect to the *standard* program semantics.

Interprocedural local-heap Shape Analysis. We develop shape analysis algorithms by abstract interpretation of our local-heap semantics. These algorithms support procedural abstraction by computing a conservative characterization of a procedure's behavior as a transformer of *local* heaps. More specifically, we develop two frameworks for interprocedural shape analysis algorithms:

- A framework for designing interprocedural shape analysis algorithms for *cutpoint-free programs* (programs in which procedure invocations never generate a cutpoint). By limiting our attention to cutpoint-free programs, we can easily harness shape abstractions designed for intraprocedural analysis, e.g., [DRS00], for interprocedural analysis. The algorithms generated by this framework are abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$. (See Chapter 3).
- A framework for designing interprocedural shape analysis algorithms for programs with arbitrary cutpoints. This framework is parameterized by the cutpoint-abstraction. Specifically, we show how we can employ *canonical abstraction* [SRW02] to handle arbitrary programs by taking special care of cutpoints in the abstraction. The algorithms generated by this framework are abstract interpretation of $\mathcal{LSL}$. (See Chapter 4).

Our frameworks are based on canonical abstraction [SRW02], and are context- and flow-sensitive with the ability to perform destructive pointer updates.

We have implemented prototypes of both frameworks, and used them to derive analyses for verifying interesting properties, including partial correctness of a recursive quicksort implementation.

Both frameworks can handle arbitrary programs, however, they are based on different approaches: The first framework verifies that a program is cutpoint-free. If it fails to do so, i.e., if it detects a possible cutpoint, it gives up on proving any other property. This allows the analysis designer to avoid designing an abstraction for cutpoints. The second framework handles programs that has procedure calls with cutpoints using a given cutpoint-abstraction.

Modular Shape Analysis for Dynamically Encapsulated Programs. We present *modular* shape analysis algorithms for a subset of heap-manipulating programs. We consider a program to be a collection of modules and develop shape (heap) analyses which treat each module separately.

Modular shape analysis is a challenging problem because shape analysis primarily concerns properties of the global-heap. Thus, it is at risk of degenerating into a whole program analysis, mainly because of aliasing, which, in general, is not constrained.

We are able to develop modular shape analysis algorithms by targeting our analyses to a certain class of "well-behaved" *dynamically encapsulated* programs. In these programs, live references (i.e., used before set) between subheaps manipulated by different modules form a tree. We found that the judicious use of aliasing in dynamically encapsulated programs make them more amenable for modular analysis.

Technically, we define a non-standard operational semantics ($\mathcal{DOS}$) which checks that program executions adhere to the dynamic encapsulation restriction while preserving procedure-local views of the program's state with respect to the standard semantics. Based on $\mathcal{DOS}$, we develop a module semantics which assigns a program-independent meaning to modules. We develop a conservative static analysis algorithm by abstract interpretation of the module semantics. Our algorithms analyze each module separately while considering only the subheaps manipulated by that module. Our algorithms modularly verify that a program is dynamically encapsulated and find shape invariants pertaining to the data structures of the analyzed modules. (See Chapter 5).

1.2 Thesis Organization

This thesis consists of three main chapters, Chapter 3, 4, and 5. In Section 1.3, we provide an informal overview of each of these chapters, and outline the connections between them. The rest of the thesis is organized as follows:

- Chapter 2 sets the scene by defining EAlgol, a simple imperative language, and defining its standard store-based semantics. It also provides a bird's eye view of the parametric framework for shape analysis of [SRW02].
- Chapter 3 presents $\mathcal{LSL}^{\text{CPF}}$, a storeless semantics for cutpoint-free programs, and develops interprocedural shape analysis algorithms for cutpoint-free programs by abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$.
- Chapter 4 presents $\mathcal{LSL}$, a storeless semantics for programs with arbitrary cutpoints, and develops interprocedural shape analysis algorithms by abstract interpretation of $\mathcal{LSL}$.
- Chapter 5 presents our modular analysis of (dynamically encapsulated) heap-manipulating programs.
- Chapter 6 reviews related work.
- Chapter 7 concludes the thesis and draws out possible directions for further work.

1.3 Overview

This section provides an informal overview of the content of this thesis. The section contains forward references to chapters that formally discuss the presented material.

1.3.1 Procedure Local-Heap Storeless Semantics

Formal semantics is concerned with rigorously specifying the meaning, or behavior, of programs [Win93]. In this section, we shortly discuss the notions of store-based semantics and storeless semantics for pointer programs. We then describe the *procedure local-heap storeless* semantics developed in this thesis.

1.3.1.1 Store-based vs. Storeless Semantics

A straightforward way to specify semantics of programs with dynamically allocated objects and pointers is by a *store-based* operational semantics. (See, e.g., [MS77, NNH99, Rey02].) In these semantics, every object is identified by its *location* (commonly referred to also as the object's *address*). A store-based semantics is very natural because it closely corresponds to concepts of the machine architecture.

In programming languages such as JAVA [GJSB05], where addresses cannot be used explicitly, any two heaps with isomorphic reachable parts are indistinguishable. (In particular, garbage cells have no significance). This allows representing states in a more abstract way than in languages in which addresses can be manipulated explicitly, e.g., in C [KR88], where (unsafe) `cast` statements from integers to pointers are used. This leads to the notion of *storeless semantics*, as described below.

A storeless semantics represents memory states as aliases between pointer access paths. More specifically, a heap is represented as a collection of objects, where each object is identified by the unique set of access paths that point to it. (cf. store-based semantics, where each object is identified by a unique, yet essentially arbitrary, location.) Note that as a result, a storeless semantics gives a canonic representation for any two heaps with isomorphic reachable parts. Also note that a storeless semantics does not give immutable identifiers to objects: Because of memory mutations, an object may be identified by different sets of access paths in different memory states.

Storeless semantics was first introduced by Jonkers [Jon81]. Jonker's original work does not handle procedure calls. Deutsch [Deu92b] defines a storeless semantics that handles procedure-calls, and uses it to develop a may-alias pointer-analysis algorithm. In this semantics, *pending access paths* (i.e., access paths starting at local variables of pending calls) are explicitly represented.

It is unfortunate that existing storeless semantics associate the entire heap with each procedure invocation: The first step in many heap-abstractions is to abstract away from specific memory addresses, e.g., [Deu92a, Deu94, RS01, SRW02, JLRS04, MYRS05, DOY06, LAIS06, MBC+07]. A storeless *concrete* semantics has already performed this step, which relieves the designer of an abstraction from having to do so. Thus, it is natural to base powerful pointer (shape) analysis algorithms on storeless semantics. However, associating the entire heap with each procedure invocation makes it difficult to design analyses which are based on a storeless semantics and support procedure and data abstraction.

Another problem with current storeless semantics is that because objects do not have immutable identifiers, it is hard to relate properties of objects before and after a call. As a result, it is hard to scale analyses based on storeless semantics to prove properties of real-life programs. By "scaling", we mean not just cost issues but also precision. In particular, even in the concrete execution, after a procedure call returns, some information about the calling context may be lost.

1.3.1.2 Procedure Local-Heaps Storeless Semantics

In this thesis, we develop storeless semantics for imperative programs with procedures that do not represent pending access paths. We say that an object is *relevant for the invocation* of a procedure when it is reachable from one of the actual parameters when the procedure is invoked. We develop storeless semantics in which procedures are invoked on *local-heaps* containing only relevant objects (and not containing irrelevant objects).

We develop two storeless procedure local-heap semantics: $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$. Both semantics are large-step (natural) operational semantics [Kah87] for single-threaded imperative programs. The two semantics differ in the way that they treat *cutpoints*, as described below.

Cutpoints

We say that an object in the local-heap of an invocation of a procedure is a *cutpoint* for that invocation when it separates the *procedure local-heap of that invocation*, i.e., the part of the heap that can be accessed by the procedure in that invocation, from the rest of the heap, which—*from the viewpoint of that procedure*—is non-accessible and immutable. (See Definition 3.3.2). More specifically, we say that an object in the procedure local-heap is a *cutpoint* for an invocation when it is reachable via a pointer-access path that (i) starts at a variable of a *pending* call and (ii) does not *traverse* the local-heap. (Thus, we do not consider an object pointed-to by a formal parameter of an invocation as one of its cutpoints.)

> **Example 1.3.1** Figure 1.1 illustrates the notions of local-heaps and cutpoints. To gain intuition, Figure 1.1 shows these notions using the familiar *store-based* [MS77, NNH99, Rey02] *small-step* [Plo81, NNH99] stack-based operational semantics.
>
> The figure depicts a memory state of a program comprised of four procedures: `main`, `foo`, `bar`, and `zoo`. The figure depicts a memory state that may occur at the entry to `zoo`. The stack of activation records is depicted on the left side of the diagram. Each activation record is labeled with the name of the procedure it is associated with. Thus, as we can see, `zoo` was invoked by `bar`; procedure `bar` was invoked by `foo`; and `foo` was invoked by the `main` procedure. The activation record at the top of the stack pertains to the *current* procedure (`zoo`). All other activation records pertain to *pending* procedure calls. Thus, for example, the access paths `z1.f1.f1`, `y9`, and `x5.f2` are pending access paths.

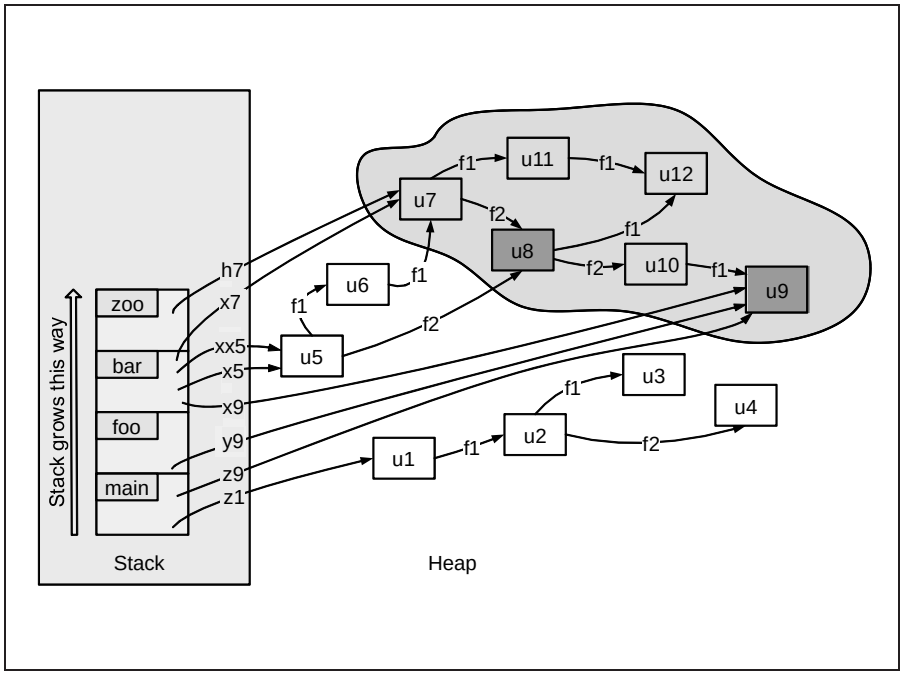


Figure 1.1: An illustration of the cutpoints for an invocation in a store-based small-step (stack-based) operational semantics at the entry to `zoo`. We assume that $h7$ is `zoo`'s formal parameter.

Heap-allocated objects are depicted as rectangles labeled with their location. The value of a pointer variable (resp. field) is depicted by an edge labeled with the name of the variable (resp. field). The shaded cloud marks the part of the heap that `zoo` can access (i.e., the part of the heap containing the relevant objects for the invocation). The cutpoints for the invocation of `zoo` ($u8$ and $u9$) are heavily shaded. Note that $u7$ is not a cutpoint although it is pointed-to by pending access paths that do not traverse through the shaded part of the heap (specifically, `x7`, `x5.f1.f1` and `xx5.f1.f1`). This is because $u7$ is also pointed-to by $h7$, `zoo`'s formal parameter.

1.3.1.3 $\mathcal{LSL}^{\text{CPF}}$: A Storeless Semantics for Cutpoint-Free Programs

$\mathcal{LSL}^{\text{CPF}}$ forbids cutpoints. We refer to a procedure invocation in which no cutpoint object exists as a *cutpoint-free invocation*. We refer to an execution of a program in which all invocations are cutpoint-free as a *cutpoint-free execution*, and to a program in which all executions are cutpoint-free as a *cutpoint-free program*.

Cutpoint-Freedom. The main idea behind the design of $\mathcal{LSL}^{\text{CPF}}$ is to restrict the "sharing patterns" occurring in procedure calls between the procedure's local-heap and the rest of the heap: $\mathcal{LSL}^{\text{CPF}}$ checks that every procedure invocation is cutpoint free. When a procedure invocation is not cutpoint-free, the semantics halts in an error state. In a sense, $\mathcal{LSL}^{\text{CPF}}$ restricts the "sharing patterns" occurring in procedure calls in such a way that the only objects that separate the procedure local-heap from the rest of the heap are the ones pointed to by the actual parameters. Stated differently, the objects pointed by the actual parameters *dominate* the part of the heap which is relevant to the invoked procedure (i.e., the callee's local-heap).

We refer to our restriction on programs as *cutpoint-freedom*. As described in Chapter 3, cutpoint-freedom greatly simplifies the handling of procedure calls in the semantics, and thus the analysis algorithm. (See also Section 1.3.2). We note that $\mathcal{LSL}^{\text{CPF}}$ does not *expect* programs to be cutpoint-free; It *checks* that they are. This makes it applicable for arbitrary programs.

Example 1.3.2 $\mathcal{LSL}^{\text{CPF}}$ can detect that the invocation of `zoo` on the memory state shown in Figure 1.1 is not cutpoint-free by inspecting only the access paths of the caller. Specifically, it would

detect that the caller (procedure `bar`) can bypass $u7$ into `zoo`'s local-heap by access paths `x5.f2` and `xx5.f2` (shown in Figure 1.1 as pointing to $u8$) and by access path `x9` (shown in Figure 1.1 as pointing to $u9$). Thus, $\mathcal{LSL}^{\text{CPF}}$ would deem this invocation to be non cutpoint-free, and abort the execution in an error state.

Storeless Semantics. Because our goal is to perform static analysis, $\mathcal{LSL}^{\text{CPF}}$ is a *storeless semantics* [Jon81]; every dynamically allocated object o is represented by the set of *access paths* that reach o. In particular, unreachable objects are not represented.

$\mathcal{LSL}^{\text{CPF}}$ differs from previous storeless semantics based on pointer-access paths [Deu92a, Ven99] in the following way: It does not represent access paths that start from variables of pending calls in the "local state" of the current procedure. This means that a procedure has a local view that only includes objects that are reachable from the procedure's parameters and, in addition, any objects that it allocates.

Observational Soundness. $\mathcal{LSL}^{\text{CPF}}$ is shown to be *observationally sound* with respect to a standard store-based semantics, i.e., *if* the program is cutpoint-free, any property that a procedure can observe according to the standard semantics, it can also observe in $\mathcal{LSL}^{\text{CPF}}$. (See Theorem 3.5.3). This allows program analyses based on an abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$ that prove properties ranging from the absence of runtime errors to partial and total correctness *with respect to the standard store-based semantics*. Furthermore, $\mathcal{LSL}^{\text{CPF}}$ is shown to be capable of *checking* cutpoint-freedom. This makes analyses based on an abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$ applicable for arbitrary programs: such an analysis can conservatively verify that the analyzed program is cutpoint-free. (See Theorem 3.5.3).

1.3.1.4 $\mathcal{LSL}$: A Storeless Semantics for Programs with Arbitrary Cutpoints

$\mathcal{LSL}$ is a procedure local-heap storeless semantics. In this respect, it is similar to $\mathcal{LSL}^{\text{CPF}}$ and differs from other existing storeless semantics [Deu92a, Ven99]. However, $\mathcal{LSL}$, in contrast to $\mathcal{LSL}^{\text{CPF}}$, allows for arbitrary cutpoints. Note that as a result, when a procedure returns some pending access path may point into the callee's local-heap, bypassing the object pointed to by the actual parameters. Determining the effect of the procedure on such access path in a storeless semantics is challenging because objects do not have unique immutable identifiers (e.g., addresses).

The main insight behind $\mathcal{LSL}$ is that the side-effects of a procedure invocation on R-values of pending access paths can be delayed to the procedure return—even though the memory cells do not have unique identifiers, e.g., locations. Intuitively, $\mathcal{LSL}$ is able to determine the effect of a procedure call by labeling every cutpoint with a unique (canonic) label when the procedure is invoked. When the procedure returns, the cutpoint-labels are used to update the caller's local-heap with the effect of the call.

Technically, $\mathcal{LSL}$ represents objects using access paths that start at the (current) procedure's local variables (in that aspect, it is similar to $\mathcal{LSL}^{\text{CPF}}$) and, *in addition*, with access paths rooted at cutpoints, which are (uniquely and immutably) labeled throughout the invocation.

$\mathcal{LSL}$ provides a context-independent representation for the cutpoints of the invocation using *cutpoint-labels*. A cutpoint-label is a set of access paths starting at formal parameters. The label of a cutpoint is the set of all access paths that start with a formal parameter (of the invoked procedure) and point-to that cutpoint when the procedure execution starts. The label of a cutpoint does not change throughout the execution of the procedure's body, even if the heap is mutated by destructive updates.

Example 1.3.3 Figure 1.2 illustrates the notion of storeless representation of procedure local-heaps using cutpoints. The figure depicts the $\mathcal{LSL}$ memory state that arises at the entry to procedure `zoo` at the memory state shown in Figure 1.1. The local-heap in Figure 1.1 has two cutpoints: $u8$ and $u9$. The cutpoint-label of the former is $cpl_8 = \{\widehat{h7.f2}\}$ and of the latter is $cpl_9 = \{h7.\widehat{f2.f2}.f1\}$. (We use the notation $\widehat{S}$ to denote that a set S of access paths is a cutpoint-label).

Every object is represented by a set of access paths starting either from the object pointed to by `h7`, `zoo`'s formal parameter (these access paths start with $h7$), or from a cutpoint (these access paths start with the appropriate cutpoint-label).[1]

[1] Allowing access paths to start at cutpoint-labels is a slight generalization of the notion of an access path. For details see Section 4.3.

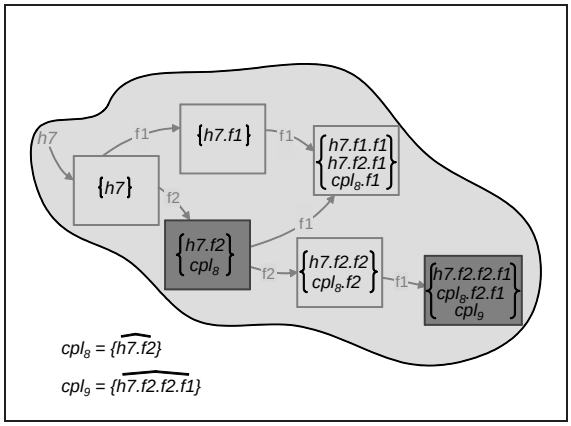


Figure 1.2: An illustration of procedure zoo's local-heap (with cutpoints) at the memory state shown in Figure 1.1 according to the $\mathcal{LSL}$ semantics. We assume that $h7$ is zoo's formal parameter.

For example, the object depicted at location $u12$ in Figure 1.1, is represented by the set of access paths $\{h7.f1.f1, h7.f2.f1, \{\widehat{h7.f2}\}.f1\}$. Access paths $h7.f1.f1$ and $h7.f2.f1$ are usual access paths starting at program variable h7. Access path $\{\widehat{h7.f2}\}.f1$ starts at the cutpoint object labeled by the (singleton) set of access paths $\{\widehat{h7.f2}\}$.

We note that Figure 1.2 uses the light gray rectangles and the light gray labeled edges connecting them only to provide intuition. The set-of-access-paths representation of the allocated objects suffices to capture the layout of the heap.

Context Independent Encoding of Sharing Patterns. The cutpoint-labels induced by a procedure call encodes the entry points into the procedure local-heap (bypassing the procedure parameters) from the rest of the heap in a context independent way: The external sharing is recorded in the same way independently of whether is due to heap sharing or stack sharing. We refer to this context independent way for recording the external sharing as the *sharing pattern* between the procedure local-heap and the rest of the heap. If we take a traditional global view of the heap, as done in Example 1.3.1, we see that cutpoints represent the external sharing. For example, in the local-heap of procedure zoo, depicted in Figure 1.1, objects $u8$ and $u9$ are cutpoints. $\mathcal{LSL}$ labels these cutpoints using the cutpoint-labels $\{\widehat{h7.f2}\}$ and $\{h7.\widehat{f2.f2}.f1\}$, respectively. Note that in this encoding, the fact that $u8$ is a cutpoint due to heap sharing and $u9$ is a cutpoint due to stack sharing, as well as the names of the variables and fields pointing to these cutpoints from outside the local-heap, is abstracted away.

Observational Equivalence. $\mathcal{LSL}$ is shown to be *observationally equivalent* to a standard store-based semantics. This allows it to be used to prove properties ranging from the absence of runtime errors to partial and total correctness *with respect to the standard store-based semantics*. In addition, it has a number of standard properties including *full abstraction* and *determinism*. (See Section 4.5).

Remark 1.3.4 *Informally, a procedure local-heap semantics replaces the more standard* call-by-reference *calling convention (where procedures are passed references to the global-heap) with a* call-by-value-result *calling convention (with local-heaps being the value* copied *when the procedure starts executing, and* restored *when it terminates). This non standard view allows to delay the propagation of the procedure's side-effects until it returns: Instead of treating the heap as a single global resource which is shared by all procedures, a local-heap semantics views the heap as a local resource of every procedure, and delays the propagation of the side effects of an invoked procedure until it returns.*

Remark 1.3.5 *The observation regarding the uniform effect of a procedure on pending access paths was already utilized in [Deu92a, LR92] for pointer analysis. We believe that our work is the first to explore the usage of this observation in semantics.*

The material pertaining to $\mathcal{LSL}^{\text{CPF}}$ is described in detail in Chapter 3. This material originally appeared in [RSY05a, RSY05b]. The material pertaining to $\mathcal{LSL}$ is described in detail in Chapter 4. This material originally appeared in [RBR$^+$05, RBR$^+$04]. Section 6.1 contrasts our work with existing works on storeless semantics.

1.3.2 Interprocedural Local-Heap Shape Analysis

Static program analysis algorithms determine statically (i.e., without the programs being actually executed) dynamic properties of programs (i.e., properties of program executions). The analysis results are valid for every possible input. Because, in general, program verification is an undecidable problem, the key idea is that of approximation, as formalized by the theory of abstract interpretation [CC79]. (For an introduction to the theory of abstract interpretation and its applications, see, e.g., [Cou00, Cou96].) Interprocedural analysis concerns the static examination of programs consisting of multiple procedures. (See Section 6.3.1 for a short review of existing approaches for interprocedural program analysis).

$\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ have been used to develop frameworks for interprocedural shape analysis. The instantiations of these frameworks are abstract interpretation algorithms of $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$, respectively. These algorithms employ the *functional* approach for interprocedural analyses [CC78, SP81, KS92, RHS95, BR01, DLS02, JLRS04]:[2] they tabulate abstractions of memory configurations before and after procedure calls. Because the instantiated algorithms abstract procedure local-heap semantics, they tabulate *local-heaps* and not whole heaps. As a result, our analyses are modular in the heap: they allow reusing the effect of a procedure at different calling contexts.

We develop two frameworks for interprocedural shape analysis:

Interprocedural shape analysis for cutpoint free programs The first framework is designed for the class of *cutpoint free* programs. It is obtained by abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$. While many programs are not cutpoint-free, we observe that a reasonable number of programs, including all examples used in [DRS00, RS01, JLRS04] are cutpoint-free, as well as many of the programs in [Deu94, SYKS03].

One of our key observations here, is that we can exploit cutpoint-freedom to construct an interprocedural shape analysis algorithm that efficiently reuses procedure summaries. More specifically, our analysis benefits not only from the tabulation of local-heaps instead of global-heaps (thus it can ignore the part of the heap not reachable from actual parameters), but also from cutpoint freedom: when analyzing a procedure call, it exploits the fact that any access path into the callee's heap must traverse through an object pointed to by a parameter. In particular, the handling of procedure returns can be greatly simplified. (See Section 4.10).

Interprocedural shape analysis for programs with cutpoints The second framework is designed for programs with arbitrary cutpoints. It is obtained by abstract interpretation of $\mathcal{LSL}$. The framework allows the analysis designer to define the expected cutpoints. The analysis becomes imprecise (and sometimes expensive) in programs in which several cutpoints are summarized together. Indeed, it is instructive to distinguish between two dimensions of heap abstractions: (i) The abstraction of the local-heap which discriminates between different kinds of aliases inside the reachable part of the heap. For example, locally-unshared objects, i.e., objects which are pointed to by one selector *from the local-heap* can be treated differently from locally-unshared objects, i.e., objects which are pointed to by two or more selectors *from the local-heap*. (ii) The abstraction of the *sharing patterns* between the local-heap and the rest of the heap (as encoded by the cutpoint-labels).

We note that while this framework can handle programs that generate arbitrary cutpoints, it is expected to work better when the number of cutpoints is bounded in any procedure invocation as it can avoid from abstracting cutpoints together.

[2] See Section 6.3.1.2 for a short review of the functional approach for interprocedural analyses.

1.3.2.1 Shape Analysis

Our analyses follow the *functional approach* for interprocedural analysis, and compute procedure summaries by tabulating (shape abstractions of the) procedure's input-output relation.

The shape abstractions employed by our analyses are obtained using the 3-valued logical framework for program analysis of [SRW02]. Thus, our interprocedural frameworks are parametric in the heap abstraction and in the concrete effects of program statements, allowing to experiment with different instances of interprocedural shape analyzers. For example, we can employ different abstractions for singly-, doubly-linked lists, and trees. Both frameworks were implemented using TVLA [LAS00]. (In Section 2.5, we summarize the framework of [SRW02]).

1.3.2.2 Experimental Evaluation

We implemented our interprocedural shape analysis algorithm and provide initial experimental results. Our empirical evaluation indicates that the analysis is precise enough to prove properties such as the absence of null dereferences, preservation of data structure invariants such as list-ness, tree-ness, and sorted-ness for iterative and recursive in the programs that we experimented with. We note that these programs have deep references into the heap and use destructive updates. We were able to verify properties that could not be automatically verified before, including the partial correctness of a recursive quicksort [Hoa61] implementation (i.e., our analysis verifies that the quicksort procedures always returns an ordered permutation of its input). We observe that in our experiments, the cost of analyzing recursive procedures is comparable to the cost of analyzing their iterative counterparts. Moreover, we found that in our experiments, the cost of analyzing many programs with procedures is smaller than the cost of analyzing the same programs with procedure bodies inlined.

Remark 1.3.6 *Abstracting local-heaps instead of global-heaps can reduce the asymptotic complexity of the interprocedural shape analysis. For example, when abstracting $\mathcal{LSL}^{\text{CPF}}$ using canonical abstraction [SRW02] for programs without global variables, the worst case time complexity of the analysis is doubly exponential in the maximum number of local variables in a procedure, instead of being doubly exponential in the overall number of local variables (as in, e.g., [RS01]).*

The material pertaining to our first framework is described in detail in Chapter 3. This material originally appeared in [RSY05a, RSY05b]. The material pertaining to our second framework is described in detail in Chapter 4. This material originally appeared in [RBR$^+$05, RBR$^+$04, RSY04]. Section 6.3 provides a bird's-eye view of existing techniques for interprocedural program analysis, concentrating on ones which have been used for interprocedural *shape* analysis, and contrasts existing interprocedural *shape* analysis algorithms with ours.

1.3.3 Modular Shape Analysis for Dynamically Encapsulated Programs

Static analysis algorithms are chosen to offer a certain trade-off between the precision of the information extracted from the program and the efficiency of the analysis algorithms [CC02]. Shape analysis is interested in very intricate properties. Thus, it usually requires quite sophisticated and rather expensive algorithms. E.g., the algorithms developed using canonical abstraction [SRW02] are doubly exponential in program size in the worst case. Such a complexity can become prohibitive for large programs.

One possible solution to the above problem is that of compositional separate static analysis of program parts where programs are analyzed by analyzing parts (such as libraries, modules, classes, procedures, etc.) separately and then by composing the analyses of these program parts to get the required information on the whole program. Indeed, modular analysis is attractive because it promises scalability and reuse. An additional advantage of modular analysis is that program parts can be analyzed with a high precision compared, e.g., to an abstraction that we can afford to use in a whole program analysis [CC02].

Modular analysis, however, is challenging when the analysis has to consider the effects of one program part on other parts. In particular, as the goal is to be able to consider the effects of a program part without reanalyzing it. Modular analysis is particularly difficult in the presence of aliasing: The behavior of a module can depend on the aliasing created by clients of the module and vice versa. Analyzing a module making worst-case assumptions about the aliasing created by clients (or vice versa) can complicate the analysis and lead to imprecise results.

Current approaches for modular shape analysis handle the challenging problem of obtaining a modular analysis in the presence of a global-heap by either restricting the program to use a bounded number of non hierarchical data structures [LKR05, WKL+06], not tracking properties of objects passed as parameters [Log03, Log04a], or breaching layers of abstraction [YSRS05].

In Chapter 5, we present a *modular* static analysis which identifies structural (shape) invariants for *parts* of heap-manipulating programs.[3] Our algorithm allows for modular analysis of hierarchical data structures without breaching layers of abstraction for programs. The main idea is to concentrate on programs that use reference parameters in a restricted way, as described below.

Instead of analyzing arbitrary programs, we restrict our attention to certain "well-behaved" programs. The main idea behind our approach is to assume (and verify) a modularly-checkable program-invariant concerning aliases of live intermodule references. More specifically, we refer to subheaps manipulated by different program modules[3] as *heap components*. The decomposition is based on the module structure of the program: Every heap component is comprised of a maximal weakly connected component of objects whose types are defined in the same module.[4] We refer to the parent of a heap component in the component tree as the *owner* of the component.

We *assume and verify* that (i) the intermodule live references between different heap components form a tree and (ii) all references to a component from its owner have the same target object. (We refer to this target object as the component's *header*).

> **Example 1.3.7** Figure 1.3(a) (b) show excerpt of a JAVA program comprised of two packages, package `RM`, implementing a simple *resource manager*, and package `RP`, implementing resources and resource pools.
>
> A resource manager has two fields pointing to *pool manager* objects. Every pool manager has one *resource pool* containing *fresh* resources and one containing *used* resources. The resource pools maintain a list of resources.
>
> Figure 1.3(c) depicts a possible memory state of this program and its component decomposition. Every allocated object is drawn as a shaded shape. The class of an object is depicted by its shape: resource managers are depicted as diamonds, pool managers are depicted as octagons, resource pools are depicted as pentagons, and resources are depicted as squares.
>
> Objects that belong to package `RM` are drawn heavily shaded. Objects that belong to package `RP` are drawn lightly shaded. Reference fields are depicted as arrows. Every heap component is circumscribed with a frame. The intercomponent references, `fresh` and `used`, pointing from subheaps manipulated by package `RM` to subheap manipulated by package RP are drawn as a wider arrow. Header objects are depicted with a double line arrow pointing to them.

Before we describe the analysis, we explain the two key issues that motivate the constraints that we place on sharing across modules. We describe these issues by posing two challenges that should be addressed by any modular analysis:

1. How can we analyze a module M without using any information about the clients of M (*i.e.*, without using information about the usage context of M)?

2. When analyzing a client module C that makes use of another module M, how do we handle *intermodule* calls from C to M using only the analysis results for module M (*i.e.*, without analyzing module M again)?

Informally, the requirement that the (live) inter-components references form a tree ensures that distinct components do not share (live) state. Furthermore, the requirement that all references to a component have to point to its header simplifies the aliasing that needs to be tracked between a component and its owner. These constraints let us deal with the two issues mentioned above in a tractable way. The restriction on sharing between components simplifies dealing with intermodule calls as they cannot have unexpected side-effects: *e.g.*, an intermodule call on one component C_1 cannot affect the state of another component C_2 that is accessible to the caller. As for the first issue, *we conservatively identify all possible input states for an intermodule call by iteratively identifying all possible components that can be generated by a module*.

[3] Our analysis uses *modules* as program parts. Intuitively, a module can be thought of as a *package* in JAVA: a collection of type-definitions and procedures that manipulate objects of these types. (See Section 5.4).

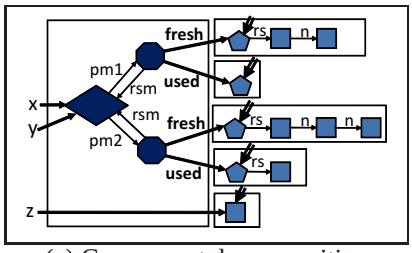
[4] Note that multiple components belonging to the same module may co-exist.

```
package RM;
  public class ResourceManager {
    // intramodule references
    private PoolManager pm1;
    private PoolManager pm2;
    ...
  }

  public class PoolManager {
   // intramodule references
   private ResourceManager rsm;

   // intermodule references
   private RPool fresh;
   private RPool used;
   ...
  }
```

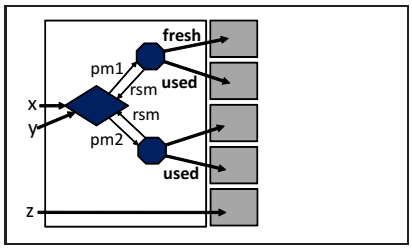
(a) Package RM

```
package RP;
  public class RPool {
    // intramodule references
    private R rs;

    @transferred: { e }
    public void release(R e){
      e.n=this.rs;
      this.rs=e;
    }

    @transferred: { }
    public R acquire(){
      R r = this.rs;
      if (r!=null) {
        this.rs=r.n;
        r.n = null;
      }
      else {
        r = new R();
      }
      return r;
    }
  }

  public class R {
   R n; ...
  }
```

(b) Package RP



(c) Component decomposition



(d) Trimmed state

Figure 1.3: A JAVA program comprised of two packages and a possible memory state of the program with its component decomposition. (a) a package implementing a resource manager. (b) a package implementing a resource pool. (c) a component decomposition of a memory state. (d) a trimmed memory state.

Our analysis employs a lightweight user specification. This specification consists of: (i) a *module specification* that partitions a program's types and procedures into modules; (ii) an annotation for every (public) procedure that indicates for every parameter whether it is intended to be "transferred" to the callee or not; these annotations are only considered in intermodule procedure calls. A component that is passed as a *transferred* parameter of an intermodule call cannot be subsequently used by the calling module (*e.g.*, to be passed as a parameter for a subsequent intermodule call). This constraint serves to directly enforce the requirement that the heap forms a tree of components.

The analysis treats each module separately: Given a module, and the user specification for the other modules it uses, our analysis tries to verify that the given module is "well-behaved" (i.e., respects its specification and the specification of the modules that it uses). If this verification is unsuccessful, the analysis gives up and reports that the module may not adhere to our constraints. Otherwise, the analysis computes invariants of the given module that hold in any "well-behaved" program containing the module. A program comprised only of successfully verified modules is guaranteed to be "well-behaved" (i.e., dynamically encapsulated). Furthermore, given the above specification, our modular analysis can automatically detect the boundaries of the heap-components and thus (conservatively) determine whether the program satisfies the constraints described above in an automatic fashion.

Technically, we define the class of *dynamically encapsulated* programs by means of a non-standard operational semantics, $\mathcal{DOS}$, which places certain restrictions on aliasing and sharing across modules. More specifically, in dynamically encapsulated programs, live references (i.e., used before set) between subheaps manipulated by different modules form a tree. We say that a program is "well behaved" if it is dynamically encapsulated.

$\mathcal{DOS}$ supports a componentized view of the memory state: It decomposes the heap into *heap components*, where a heap component is a subheap. The decomposition is based on the module structure of the program as described above. $\mathcal{DOS}$ represents the heap as an *evolving and changing* tree of heap components.

Based on $\mathcal{DOS}$, we define a concrete module semantics which assigns a program-independent meaning to every module. The module semantics is obtained using a novel *trimming abstraction* which abstracts away heap components that are not manipulated by the analyzed module. We develop conservative static analysis algorithms by abstract interpretation of the module semantics.

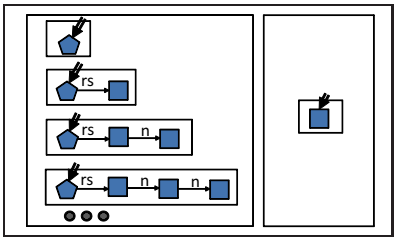


Figure 1.4: Module invariants of package RP. (Packages RP is shown in Figure 1.3(b)).

Example 1.3.8 Figure 1.3(d) depicts the trimmed memory state resulting from applying the trimming abstraction to the memory state depicted in Figure 1.3(d). The trimmed memory state maintains the information regarding the internal structure of the resource manager and that the intermodule references leaving the component are not aliased. However, it loses the information regarding the internal structure of the heap components that are managed by package RP, which are depicted as shadowed rectangles.

We then apply a *bounded* conservative abstraction of trimmed memory states. Rather than providing a new intraprocedural abstraction, we show how to *lift* existing *intra*procedural shape analyses, e.g., [SRW02, MYRS05, DOY06, LAIS06], to obtain a modular shape abstraction. The common aspect of [MYRS05, DOY06, LAIS06] is that they provide information regarding *domination from variables* which is required by our analysis. Thus, our analysis is parametric in the abstraction of trimmed memory states and can use different (bounded) abstractions when analyzing different modules.

Example 1.3.9 The specification of the resource pool manager, shown in Figure 1.3(b), ensures that the resource pool's client gets ownership of an acquired resource and transfers the ownership of a released resource to the pool. As a result, a client cannot use an acquired resource after its release, e.g., it cannot release it into another pool.

Figure 1.4 depicts the concrete module invariant of package RP, i.e., all the resource components and all the resource pool components that can arise in any dynamically encapsulated program. The invariant contains component comprised of a single resource and components comprised of pools with an acyclic and unshared list of nodes (of arbitrary size), and *only* these pools. For this module, our analysis computes a conservative description of the resource invariant. In particular, it shows that the resource lists are always unshared and acyclic.

This material is described in detail in Chapter 5. It originally appeared in [RPHR$^+$07, RPHR$^+$06]. The idea of a componentized heap abstraction was also used in [RRSY06] for static analysis of parametric data structures. (See Section 6.3.5). In Section 6.4, we contrast our work with existing approaches for modular verification of heap manipulating programs.

Chapter 2

Preliminaries

In this chapter, we introduce EAlgol, a simple imperative procedural language, and define for it two observationally equivalent operational semantics: $\mathcal{GSB}$ (for *Global-heap Store-Based*), which we consider to be the *standard semantics* of EAlgol, and $\mathcal{LSB}$ (for *Local-heap Store-Based*). We also summarize the parametric framework for shape analysis via 3-valued logic of [SRW02] by using it to sketch an *intra*procedural shape analysis algorithm for programs manipulating singly linked lists.

$\mathcal{GSB}$ and $\mathcal{LSB}$ are large-step (natural) [Kah87, NNH99] store-based [MS77, NNH99, Rey02] semantics. However, while $\mathcal{GSB}$ is a global-heap semantics, i.e., invoked procedures execute on a heap containing all allocated objects, $\mathcal{LSB}$ is a local-heap semantics, i.e., procedures are invoked on a *local-heap* containing only the objects that are reachable from the actual parameters. (We refer to these objects as the *relevant objects for the invocation*.) Nevertheless, a program cannot tell the two semantics apart, i.e., a program cannot observe whether it is being executed according to the $\mathcal{GSB}$ semantics or according to the $\mathcal{LSB}$ semantics.

Outline. In Section 2.1, we define the syntax of EAlgol. In Sections 2.2 and 2.3, we define the $\mathcal{GSB}$ semantics and the $\mathcal{LSB}$ semantics, respectively. In Section 2.4, we show that these two semantics are observationally equivalent. In Section 2.5, we review the use of 3-valued logic for shape analysis.

2.1 The Syntax of EAlgol

EAlgol is a simple imperative procedural language. Programs in EAlgol consist of a collection of procedures including a `main` procedure. The programmer can also define her own types (à la `C` structs) and refer to heap-allocated objects of these types using pointer variables. Parameters are passed by value. Formal parameters cannot be assigned to. Procedures return a value by assigning it to a designated variable `ret`. Constants are either integers or the designated value `null`.

The syntax of EAlgol is defined in Figure 2.1. The notation $\bar{z}$ denotes a sequence of z's. We define the syntactic domains $x, y \in \mathcal{V}$, $f \in \mathcal{F}$, $p \in \mathit{FuncId}$, $t \in \mathcal{T}$, and $lb \in \mathit{Labels}$ of variables, field names, procedure identifiers, type names, and program-labels, respectively. For a procedure p, V_p denotes the set of its local variables and F_p denotes the set of its formal parameters. We assume $F_p \subseteq V_p$ and that all the variables in $V_p \setminus F_p$, with the exception of the specially designated variable `ret`, are declared at the beginning of p's declaration. For simplicity, we assume that (i) variables, fields, procedures, types, and program-labels have unique identifiers in every program; (ii) formal parameters *cannot* be assigned to; (iii) a program has only local variables; and (iv) a program manipulates only pointer-valued fields and variables.[1]

To simplify notation, we assume that we work with a fixed arbitrary program P.

[1] These assumptions are made to simplify the presentation. In principle, different ones could be used with minor effects on the capabilities of our approach. For clarity, our example programs do not adhere to these restrictions.

P	$\in$	*prog*	$::=$	$\overline{\textit{rcdecl}}\ \overline{\textit{prdecl}}$
		rcdecl	$::=$	`record` $t := \{\ \overline{\textit{tname}\ f}\ \}$
		prdecl	$::=$	*tname* $p(\overline{\text{tname}\ x})\ := \overline{\textit{vdecl}}\ st$
		vdecl	$::=$	*tname* $\overline{x}$
		tname	$::=$	`int` $\mid\ \check{}\,t$
st	$\in$	*stms*	$::=$	$x{=}e \mid x{=}y.f \mid x.f{=}e \mid x = \texttt{alloc}\ t \mid$
				$y{=}p(\overline{x}) \mid$
				$lb\colon st \mid st\,;st \mid \texttt{while}\ (\textit{cnd})\ \texttt{do}\ st\ \texttt{od} \mid$
				$\texttt{if}\ (\textit{cnd})\ \texttt{then}\ st\ \texttt{fi} \mid \texttt{if}\ (\textit{cnd})\ \texttt{then}\ st\ \texttt{else}\ st\ \texttt{fi}$
e	$\in$	*exp*	$::=$	$c \mid x \mid e \otimes e$
		$\otimes$	$::=$	$+ \mid - \mid * \mid /$
cnd	$\in$	*cond*	$::=$	$x \bowtie c \mid x \bowtie y$
		$\bowtie$	$::=$	$= \mid \neq \mid < \mid \leq$
c	$\in$	*const*	$::=$	`null` $\mid$ `n`

Figure 2.1: Syntax of EAlgol. $\check{}\,t$ denotes the type of pointers to type t.

2.1.1 Running Example

Figure 2.2(a) shows a simple list-manipulating program in EAlgol, which we use as a running example in this chapter. (Figure 2.2(b) provides, for reference, the same program written in JAVA).

The program defines type `List` of singly-linked lists. Its `main` procedure allocates three unshared, disjoint, acyclic singly-linked lists (by invoking procedure `create3` three times) and splices them together (by invoking procedure `splice` two times).

Procedure `create3` allocates three list nodes; connects them into an acyclic list; and returns the resulting list as its return value.

Procedure `splice` gets two lists arguments, `p` and `q`, and recursively splices them together using destructive updates. More specifically, it generates a list comprised of the nodes in `p`'s list and in `q`'s list where the ith node in `p`'s list is followed by the ith node in `q`'s list. (In case one of the lists has more elements than the other, then these elements are placed at the tail of the generated list). The behavior of procedure `splice` depends on the value of `p`, its first argument: If `p` has a null value, then the execution of `splice` terminates and the list pointed to by `q`, `splice`'s second argument, is returned as the return value. If `p` has a non-null value, then `splice` stores a reference to `p`'s tail in variable `pn` and disconnects the head of `p`'s list (pointed to by `p`) from its tail (pointed to by `pn`). It then recursively invokes itself passing `q` as the *first* parameter and `pn` as the *second* parameter. When the recursive call terminates, `splice` stores the returned list in variable `r`; connects the node pointed to by `p` to the head of `r`'s list; and returns the resulting list.

2.1.2 Memory Deallocation

In this dissertation, we do not handle explicit memory deallocation. (Thus, EAlgol does not have a `free` or `dealloc` statement). Instead, we make the simplifying assumption that there is an unbounded amount of memory and thus memory allocation never fails. (Alternatively, one can think of a programming language with garbage collection).

We note that our analyses (conservatively) identify when a program leaks memory, i.e., when an object (or a set of objects) becomes unreachable from any program variable. This allows to use our analysis to discover opportunities for compile time garbage collection [Bar77] for *imperative* languages with destructive updates, in the spirit of [SYKS05].

2.2 $\mathcal{GSB}$: A Global-Heap Store-Based Semantics

In this section, we define the $\mathcal{GSB}$ (for *Global-heap Store-Based*) semantics. $\mathcal{GSB}$ is a large-step (natural) [Kah87, NNH99] store-based [MS77, NNH99, Rey02] semantics. It is a global-heap semantics in the following sense:

```
record List := {
   ˇList n;
   int data
}

ˇList create3(int k) :=
   ˇList t1,t2,t3;
   t1 = alloc List;
   t1.data = k;
   t2 = alloc List;
   t2.data = k+1;
   t3 = alloc List;
   t3.data = k+2;
   t1.n = t2;
   t2.n = t3;
   ret = t1


ˇList splice(ˇList p, ˇList q) :=
   ˇList w, pn, r;
   w = q;
   if (p != null) then
      pn = p.n;
      p.n = null;
      r = splice(q,pn);
      p.n = r;
      w = p
   fi;
   r = null;
   ret = w



int main():=
   ˇList x,y,z,t,s;
   x = create3(1);
   y = create3(4);
   z = create3(7);
   t = splice(x,y);
   s = splice(y,z);
   ret = 0
```

(a) The running example written in EAlgol.

```
public class List{
   List n = null;
   int data;


   static public List create3(int k) {
      List t1, t2, t3;
      List t1 = new List();
      t1.data = k;
      List t2 = new List();
      t2.data = k+1;
      List t3 = new List();
      t3.data = k+2;
      t1.n = t2;
      t2.n = t3;
      return t1;
   }

   public static List splice(List p, List q) {
      List w, pn, r;
      w = q;
      if (p != null) {
         pn = p.n;
         p.n = null;
         r = splice(q, pn);
         p.n = r;
         w = p;
      }
      r = null;
      return w;
   }
}


public class Main{
   public static void main(String[] argv) {
      List x = create3(1);
      List y = create3(4);
      List z = create3(7);
      List t = splice(x, y);
      List s = splice(y, z);
   }
}
```

(b) The running example written in JAVA.

Figure 2.2: The running example (a) in EAlgol and (b) in JAVA.

l	$\in$	Loc	Locations
v	$\in$	$Val = Loc \cup \{\mathit{null}\}$	Values
ρ	$\in$	$\mathit{Env}_p = V_p \rightarrow Val$	Environments
h	$\in$	$\mathit{Heap}_G = Loc \times \mathcal{F} \hookrightarrow Val$	
$s^p_G, s_G, \langle L, \rho, h\rangle$	$\in$	$\mathcal{S}^p_G = 2^{Loc} \times \mathit{Env}_p \times \mathit{Heap}_G$	Memory states

Figure 2.3: Semantic domains of the $\mathcal{GSB}$ semantics.

Invoked procedures execute on a heap containing all allocated objects.

2.2.1 Simplifying Assumptions

For simplicity, the semantics tracks only pointer values and assumes that every pointer-valued field or variable is assigned `null` before being assigned a new value.[2] In addition, we assume that before a procedure terminates it assigns a `null` value to every pointer variable $v \in V_p \setminus (F_p \cup \{\texttt{ret}\})$.[3] In the rest of the thesis, we assume these simplifying assumptions in every semantics that we define.

2.2.2 Memory States

Figure 2.3 defines the semantic domains and the meta-variables ranging over them. Loc is an unbounded set of memory locations. A *memory state* for a procedure p, $s^p_G \in \mathcal{S}^p_G$, keeps track of the allocated memory locations, L, an environment mapping p's local variables to values, ρ, and a mapping from fields of *allocated* locations to values, h.[4] Due to our simplifying assumptions, a value is either a memory location or *null*.

2.2.3 Operational Semantics

The meaning of statements of a procedure p is described by a transition relation $\overset{GSB}{\rightsquigarrow} \subseteq (\mathcal{S}^p_G \times \mathit{stms}) \times \mathcal{S}^p_G$. Figure 2.4 shows the *axioms* for pointer assignments. The *inference rule* for procedure calls is given in Figure 2.5. Note that $\mathcal{GSB}$ treats the heap as a global resource which is *shared* by all procedures. Specifically, (i) $h_e = h_c$, i.e., the callee starts executing from the caller's global-heap at the call-site, and (ii) $h_r = h_x$, i.e., the caller continues its execution from the global-heap at the exit state of the callee.

All other statements are handled as usual using a two-level store semantics for pointer languages. (See, e.g., [MS77, NNH99, Rey02]. Also, see Section B.1.)

> **Example 2.2.1** Figure 2.6 depicts four memory states that may arise during the first call to `splice` according to the $\mathcal{GSB}$ semantics. Allocated locations are depicted as rectangles labeled by the location name. The value of a non null-valued (pointer) variable is depicted as an arrow from the variable name to the memory location it points-to.
>
> Figure 2.6($s^{c\,2.6}_G$) depicts a memory state that may arise at the call site of the first invocation of `splice`. Figure 2.6($s^{e\,2.6}_G$) depicts the resulting entry memory state. The heap of the entry memory state is identical to the one of the call state. The environment of `splice` at the entry memory state maps the formal parameters `p` and `q` to the values of the actual parameters `x` and `y` at the call state, respectively. The execution of `splice`'s body ends with `w` pointing to the head of the spliced list. Figure 2.6($s^{x\,2.6}_G$) depicts the exit memory state. Figure 2.6($s^{r\,2.6}_G$) depicts the resulting return memory state. Note that (i) the heap of the return state is identical to the one at `splice`'s exit state and (ii) the environment of the return state is as in the call state, except that the return value is assigned to `t`.

[2] For simplicity, we assume that the same variable cannot appear both in the left-hand-side and in the right-hand-side of an assignment.

[3] The latter assumption simplifies the definition of our semantics as it ensures that every object which is reachable by the callee at the exit state is also reachable by the caller at the return state. However, this assumption, as well as the other simplifying assumptions made in this section, are not required; in principle, different assumptions could be used with minor effects on the capabilities of our approach. For clarity, our example programs do not adhere to these restrictions.

[4] We omit the p superscript when p is clear from context.

$$
\begin{array}{lll}
[\text{NULLIFY}] & \langle \mathtt{x} = \mathtt{null}, \langle L, \rho, h\rangle\rangle \overset{GSB}{\rightsquigarrow} \langle L, \rho[x \mapsto \mathit{null}], h\rangle & \\
[\text{COPYVAR}] & \langle \mathtt{x} = \mathtt{y}, \langle L, \rho, h\rangle\rangle \overset{GSB}{\rightsquigarrow} \langle L, \rho[x \mapsto \rho(y)], h\rangle & \\
[\text{DEREF}] & \langle \mathtt{x} = \mathtt{y.f}, \langle L, \rho, h\rangle\rangle \overset{GSB}{\rightsquigarrow} \langle L, \rho[x \mapsto h(\rho(y), f)], h\rangle & \rho(\mathrm{y}) \neq \mathit{null} \\
[\text{SETNULL}] & \langle \mathtt{x.f} = \mathtt{null}, \langle L, \rho, h\rangle\rangle \overset{GSB}{\rightsquigarrow} \langle L, \rho, h[(\rho(x), f) \mapsto \mathit{null}]\rangle & \rho(\mathrm{x}) \neq \mathit{null} \\
[\text{SETVAR}] & \langle \mathtt{x.f} = \mathtt{y}, \langle L, \rho, h\rangle\rangle \overset{GSB}{\rightsquigarrow} \langle L, \rho, h[(\rho(x), f) \mapsto \rho(y)]\rangle & \rho(\mathrm{x}) \neq \mathit{null} \\
[\text{ALLOC}] & \langle \mathtt{x} = \mathtt{alloc}\ t, \langle L, \rho, h\rangle\rangle \overset{GSB}{\rightsquigarrow} \langle L \cup \{l_{new}\}, \rho[x \mapsto l_{new}], h \cup I(l_{new})\rangle & \mathrm{l}_{new} \notin \mathrm{L} \\
 & \quad \text{where } I(l_{new}) = [l_{new} \mapsto \lambda f \in \mathcal{F}.\mathit{null}] &
\end{array}
$$

Figure 2.4: Axioms for pointer assignments in the $\mathcal{GSB}$ semantics. I initializes all pointer fields at l to *null*.

$$
\frac{\langle \text{body of } p, s^e_G\rangle \overset{GSB}{\rightsquigarrow} s^x_G}{\langle y = p(x_1, \ldots, x_k), s^c_G\rangle \overset{GSB}{\rightsquigarrow} s^r_G}
$$

where

$$
\begin{array}{lll}
L_e = L_c & & L_r = L_x \\
\rho_e = \lambda v. \begin{cases} \rho_c(x_i) & v = z_i \\ \mathit{null} & \mathit{otherwise} \end{cases} & & \rho_r = \rho_c[y \mapsto \rho_x(ret)] \\
h_e = h_c & & h_r = h_x
\end{array}
$$

Figure 2.5: Inference rule for procedure invocation in the $\mathcal{GSB}$ semantics, assuming the formal variables of p are $z_1, \ldots, z_k$ and that p's return value is a pointer. $s^c_G = \langle L_c, \rho_c, h_c\rangle$ is the call-state, $s^e_G = \langle L_e, \rho_e, h_e\rangle$ is the entry-state, $s^x_G = \langle L_x, \rho_x, h_x\rangle$ is the exit-state, and $s^r_G = \langle L_r, \rho_r, h_r\rangle$ is the resulting return-state.

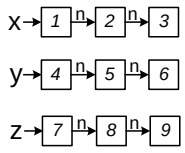
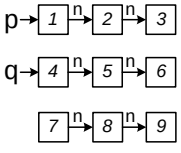
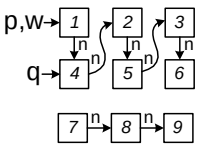
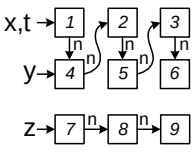
call `splice(x,y)`	enter `splice(p,q)`	exit `splice(p,q)`	return `t=splice(x,y)`
			
$(s^{c\,2.6}_G)$	$(s^{e\,2.6}_G)$	$(s^{x\,2.6}_G)$	$(s^{r\,2.6}_G)$

Figure 2.6: Concrete states for the invocation `t = splice(x, y)` in the running example according to the $\mathcal{GSB}$ semantics.

$$\frac{\langle \text{body of } p, s_L^e\rangle \overset{LSB}{\rightsquigarrow} s_L^x}{\langle y = p(x_1,\ldots,x_k), s_L^c\rangle \overset{LSB}{\rightsquigarrow} s_L^r}$$

where

$$L_e = L_c \qquad L_r = L_x$$

$$\rho_e = \lambda v.\begin{cases} \rho_c(x_i) & v = z_i \\ \textit{null} & \textit{otherwise}\end{cases} \qquad \rho_r = \rho_c[y \mapsto \rho_x(ret)]$$

$$h_e = h_c|_{L_{rel}} \qquad h_r = h_c|_{L_c \setminus L_{rel}} \cup h_x$$

$$L_{rel} = \textit{lfp}\ \lambda X.(X \cup L_{params} \cup \{l \in L_c \mid l' \in X, f \in \mathcal{F}, l = h_c(l', f)\})$$
$$L_{params} = \{\rho_c(x_i) \in L_c \mid 1 \le i \le k\}$$

Figure 2.7: Inference rule for procedure invocation in the $\mathcal{LSB}$ semantics, assuming the formal variables of p are $z_1,\ldots,z_k$ and that p's return value is a pointer. $s_L^c = \langle L_c, \rho_c, h_c\rangle$ is the call-state, $s_L^e = \langle L_e, \rho_e, h_e\rangle$ is the entry-state, $s_L^x = \langle L_x, \rho_x, h_x\rangle$ is the exit-state, and $s_L^r = \langle L_r, \rho_r, h_r\rangle$ is the resulting return-state. L_{params} is the set of locations pointed to by the actual parameters at the call state. L_{rel} is the set of relevant objects for the invocation, i.e., the locations which are reachable from L_{params} when the procedure is invoked. The operations $\cdot|_\cdot$, $\cdot \setminus \cdot$, and $\textit{lfp}\,\cdot$ are function restriction, set difference, and the least fixed point operator, respectively. (See Section A.1 for a formal definition of these operations).

2.3 $\mathcal{LSB}$: A Localized-Heap Store-Based Semantics

In this section, we define the $\mathcal{LSB}$ semantics. $\mathcal{LSB}$ is a large-step (natural) [Kah87, NNH99] store-based [MS77, NNH99, Rey02] semantics. It is a local-heap semantics in the following sense: Procedures are invoked on a *local-heap* containing only the objects that are reachable from the actual parameters of the invocation. We refer to these objects as the *relevant objects for the invocation*.[5]

2.3.1 Memory States

$\mathcal{LSB}$ uses the same semantic domains as the $\mathcal{GSB}$ semantics (see Figure 2.3). For clarity, we denote $\mathcal{LSB}$'s domain of *memory states* for a procedure p by $s_L^p \in \mathcal{S}_L^p$.[4]

2.3.2 Operational Semantics

The meaning of statements of a procedure p is described by a transition relation $\overset{LSB}{\rightsquigarrow} \subseteq (\mathcal{S}_L^p \times \textit{stms}) \times \mathcal{S}_L^p$. The meaning of intraprocedural statements in $\mathcal{LSB}$ is the same as in $\mathcal{GSB}$ (see Figure 2.4). The *inference rule* for procedure calls is given in Figure 2.7.

Example 2.3.1 Figure 2.8 depicts four memory states that may arise during the first call to `splice` according to the $\mathcal{LSB}$ semantics, using the same graphical conventions introduced in Example 2.2.1. Figure 2.8($s_L^{c\,2.8}$) depicts the call memory state; Figure 2.8($s_L^{e\,2.8}$) depicts the entry memory state; Figure 2.8($s_L^{x\,2.8}$) depicts the exit memory state; and Figure 2.8($s_L^{r\,2.8}$) depicts the return memory state.

The heap of the entry state is the restriction of the heap of the call state on the set of relevant objects for the invocation. Specifically, the heap of the entry state does not contain the list pointed to by `z` at the call state. (We only draw objects which are in the domain of the heap.) The heap of the return state is comprised of `splice`'s heap at the exit state combined with the list pointed by `z` at the call state. (Note that `z`'s list could not have affected `splice`'s behavior in its first invocation nor been affected by it). The environment at the return state is as in the call-site, except that the return value is assigned to `t`.

[5] We remind the reader that we keep making the simplifying assumptions listed in Section 2.2.1.

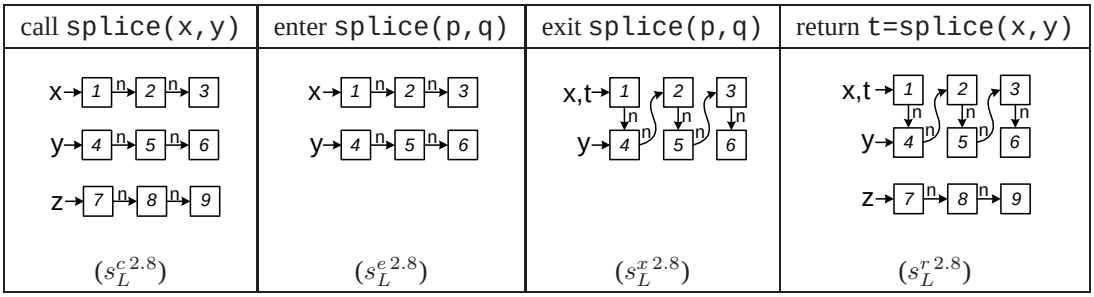


Figure 2.8: Concrete states for the invocation `t = splice(x, y)` in the running example according to the $\mathcal{LSB}$ semantics.

Remark 2.3.2 *It is interesting to note that the rule for combining the heap of the call state and the heap of the exit state is rather simple because (i) the relevant objects for the invocation are* identified using the same location names *in the call state of the caller and in the exit state of the callee; and (ii) having $L_e = L_c$ ensures that a procedure cannot allocate locations that are used to identify objects that were allocated before the procedure was invoked, and which were irrelevant for the invocation. We note that the use of a "global" set to record the locations of allocated objects is just a mechanism which ensures that locations of objects which are irrelevant for the invocation are not used. Other mechanisms that achieve this goal could have been used instead. Specifically, the local-heap of a procedure contains only the objects in the domain of its heap.*

2.4 Observational Equivalence between $\mathcal{LSB}$ and $\mathcal{GSB}$

In this section, we introduce the notion of observable properties and show that, with respect to these properties, the $\mathcal{GSB}$ semantics and the $\mathcal{LSB}$ semantics are *observationally equivalent*.

2.4.1 Observable Properties

In this section, we introduce access paths, which are the only means by which a program can observe a state.

Definition 2.4.1 (Field paths) *A **field path** $\delta \in \Delta = \mathcal{F}^*$ is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by ϵ.*

Definition 2.4.2 (Access paths) *An **access path** $\alpha = \langle x, \delta\rangle \in \mathcal{V} \times \Delta$ is a pair consisting of a local variable $x \in \mathcal{V}$ and a field path $\delta \in \Delta$. An access path $\langle x, \delta\rangle$ is an **access path of a procedure** p when x is a local variable of p. $AccPath_p = V_p \times \Delta$ denotes the (infinite) set of all access paths of procedure p (i.e., the set of all access paths starting at a local variable of p). $AccPath_P$ denotes the union of all access paths of all procedures in a program P.*

Apart from the above formal definitions, we will sometimes use notations of the form `x.n.n` for access paths, because their syntax is familiar from a number of programming languages, where it denotes a sequence of field dereferences. Because states and access paths are always associated with a (unique) procedure p, in the rest of the thesis we omit p whenever it is clear from the context.

Definition 2.4.3 *Given a heap $h \in \text{Heap}_G = Loc \times \mathcal{F} \hookrightarrow \text{Val}$, the **extension of** h **to field paths**, denoted by $\hat{h}$, is the total function*

$$\hat{h}\colon \text{Val} \times \Delta \to \text{Val} \textit{ such that}$$
$$\hat{h}(v,\delta) = \begin{cases} v & \text{if } \delta = \epsilon \\ \hat{h}(h(v,f),\delta') & \text{if } \delta = f\delta',\ \langle v, f\rangle \in dom(h) \\ \textit{null} & \text{otherwise} \end{cases}$$

Definition 2.4.4 (Value of Access paths in $\mathcal{GSB}$) *The value of an access path $\alpha = \langle x, \delta\rangle$ according to the $\mathcal{GSB}$ semantics in state $\langle L, \rho, h\rangle$, denoted by $[\![\alpha]\!]_{GSB}\langle L, \rho, h\rangle$, is $\hat{h}(\rho(x), \delta)$.*

Definition 2.4.5 (Value of Access paths in $\mathcal{LSB}$) *The value of an access path $\alpha = \langle x, \delta \rangle$ according to the $\mathcal{LSB}$ semantics in state $\langle L, \rho, h \rangle$, denoted by $[\![\alpha]\!]_{LSB}\langle L, \rho, h \rangle$, is $\hat{h}(\rho(x), \delta)$.*

Note that the value of an access path that traverses a *null*-valued field is defined to be *null*. This definition simplifies the notion of observational equivalence. Alternatively, we could have defined the value of such a path to be $\perp$. We note that both the $\mathcal{GSB}$ semantics and the $\mathcal{LSB}$ semantics check that a null-dereference is not performed (see the side-conditions of inference rules [DEREF], [SETNULL], and [SETVAR] shown in Figure 2.4).

Definition 2.4.6 (Access-path equality in $\mathcal{GSB}$) *Access paths α and β are **equal** according to the $\mathcal{GSB}$ semantics in a given state s_G, denoted by $[\![\alpha = \beta]\!]_{GSB}(s_G)$, if they have the same value in that state, i.e., $[\![\alpha]\!]_{GSB}(s_G) = [\![\beta]\!]_{GSB}(s_G)$. An access path is **equal to** $\mathbf{null}$, denoted by $[\![\alpha = \texttt{null}]\!]_{GSB}(s_G)$, if $[\![\alpha]\!]_{GSB}(s_G) = null$.*

Definition 2.4.7 (Access-path equality in $\mathcal{LSB}$) *Access paths α and β are **equal** according to the $\mathcal{LSB}$ semantics in a given state s_L, denoted by $[\![\alpha = \beta]\!]_{LSB}(s_L)$, if they have the same value in that state, i.e., $[\![\alpha]\!]_{LSB}(s_L) = [\![\beta]\!]_{GSB}(s_L)$. An access path is **equal to** $\mathbf{null}$, denoted by $[\![\alpha = \texttt{null}]\!]_{LSB}(s_L)$, if $[\![\alpha]\!]_{LSB}(s_L) = null$*

2.4.1.1 Pending Access Paths

Our semantics are natural semantics; the stack of activation records is maintained implicitly. However, we need the notion of an access path that starts at a variable of a pending call (i.e., not the current call). In a small-step semantics, this would be an access path that starts at a variable allocated in the activation record of a pending call. We use the term a *pending variable* for a local variable of a pending call, and a *pending access path* for an access path that starts at a pending variable. When we wish to emphasize that a variable (resp. access path) is of the current call, we use the term a *current variable* (resp. a *current access path*). Note that a program cannot observe the values of pending access paths.

For example, at the entry to the first invocation of `splice` the variables `x`, `y`, and `z` are pending variables. The access paths `x.n` and `z.n.n` are pending access paths. The only non null-valued current variables are `p` and `q`. The only non null-valued current access paths are `p`, `p.n`, `p.n.n`, `q`, `q.n`, and `q.n.n`. (See also Example 1.3.1).

2.4.2 Observational Equivalence

The only means by which a program can observe a state is by evaluating equality between access paths or checking whether their value is null. Thus, it cannot observe location names nor parts of the heap it cannot reach. This limitation on the possible observations that a program can do allows us to establish that the $\mathcal{LSB}$ semantics is observationally equivalent to the $\mathcal{GSB}$ semantics.

Definition 2.4.8 (Observational equivalence) *Let p be a procedure. The states $s_L \in \mathcal{S}_L^p$ and $s_G \in \mathcal{S}_G^p$ are **observationally equivalent**, denoted by $s_L \cong s_G$, if for all $\alpha, \beta, \gamma \in AccPath_p$,*

(i) $[\![\alpha = \beta]\!]_{LSB}(s_L) \Leftrightarrow [\![\alpha = \beta]\!]_{GSB}(s_G)$, *and*

(ii) $[\![\gamma = \texttt{null}]\!]_{LSB}(s_L) \Leftrightarrow [\![\gamma = \texttt{null}]\!]_{GSB}(s_G)$.

We also define observational equivalence between states in $\mathcal{LSB}$ (respectively states in $\mathcal{GSB}$) in the same way.

We denote by $s_L \ncong s_G$ that two states $s_L \in \mathcal{S}_L^p$ and $s_G \in \mathcal{S}_G^p$ are *not* observationally equivalent. We use a similar notation to denote that two states in $\mathcal{LSB}$ (respectively in $\mathcal{GSB}$) are not observationally equivalent.

> **Example 2.4.9** The two $\mathcal{LSB}$ memory states ${s_L^1}^{2.9}$ and ${s_L^2}^{2.9}$, depicted in Figure 2.9, are observationally equivalent: Every pair of access paths which are aliased in one memory state are also alias in the other one. Note, however, that the two memory states are comprised of different locations.
>
> The $\mathcal{LSB}$ memory state ${s_L^3}^{2.9}$, depicted in Figure 2.9, is not observationally equivalent to either ${s_L^1}^{2.9}$ or ${s_L^2}^{2.9}$: Note that while $[\![x = x.n]\!]_{LSB}({s_L^3}^{2.9})$, neither $[\![x = x.n]\!]_{LSB}({s_L^1}^{2.9})$ holds nor $[\![x = x.n]\!]_{LSB}({s_L^2}^{2.9})$.

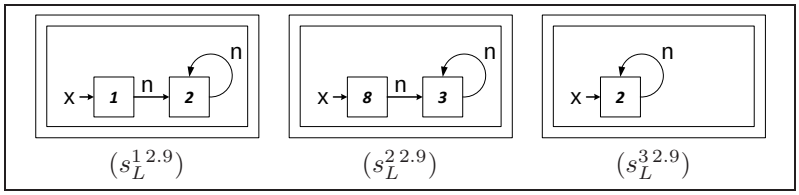


Figure 2.9: An example for observationally equivalent and non observationally equivalent memory states: $s_L^{1\,2.9} \cong s_L^{2\,2.9}$, but $s_L^{1\,2.9} \ncong s_L^{3\,2.9}$ and $s_L^{2\,2.9} \ncong s_L^{3\,2.9}$.

The following theorem states that $\mathcal{LSB}$ is equivalent to $\mathcal{GSB}$, in the sense that both behave equivalently with respect to termination (i.e., executing a statement in a given state terminates if and only if it terminates when executed in any state which is observationally equivalent to it), and that execution of statements preserves observational equivalence.

Theorem 2.4.10 (Observational equivalence between $\mathcal{LSB}$ and $\mathcal{GSB}$) *Let p be a procedure. Let $s_L \in \mathcal{S}_L^p$ and $s_G \in \mathcal{S}_G^p$ be observationally equivalent states, i.e., $s_L \cong s_G$. Let st be an arbitrary statement in p. The following holds:*

$$\langle st, s_L \rangle \overset{LSB}{\rightsquigarrow} s'_L \iff \langle st, s_G \rangle \overset{GSB}{\rightsquigarrow} s'_G.$$

Furthermore, $s'_L \cong s'_G$.

Sketch of Proof: The proof is done by induction on the shape of the derivation trees. Specifically, we establish that (i) the two derivation trees are isomorphic and that (ii) every memory state $\sigma_L \in \mathcal{S}_L^p$ is isomorphic (i.e., equivalent up to location renaming) to its matching state $s_G \in \mathcal{S}_G^p$ when the heap of the latter is projected on the set of objects that belong to the local-heap of the currently executing procedure (i.e., on the set containing the locations of the relevant objects for the invocation and of the objects that were allocated after the current procedure was invoked).

It is interesting to note that despite the fact that $\mathcal{LSB}$'s memory allocator allocates locations in a non-deterministic manner, the same program always yields observationally equivalent memory states. It follows immediately that the $\mathcal{GSB}$ semantics has the same property.

Theorem 2.4.11 ($\mathcal{LSB}$ is deterministic up to location renaming) *Let p be a procedure. Let $s_L^1 \in \mathcal{S}_L^p$ and $s_L^2 \in \mathcal{S}_L^p$ be observationally equivalent states, i.e., $s_L^1 \cong s_L^2$. Let st be an arbitrary statement in p. The following holds:*

$$\langle st, s_L^1 \rangle \overset{LSB}{\rightsquigarrow} s_L^{1'} \iff \langle st, s_L^2 \rangle \overset{LSB}{\rightsquigarrow} s_L^{2'}.$$

Furthermore, $s_L^{1'} \cong s_L^{2'}$.

Sketch of Proof: The proof is done by induction on the shape of the derivation trees. Specifically, we establish that the two trees are isomorphic and that every pair of matching states are isomorphic.

2.5 A Primer on Parametric Shape Analysis via 3-Valued Logic

Our shape analysis algorithms are expressed in terms of the 3-valued-logic framework for program analysis of [SRW02] and implemented using its realization in the TVLA framework for Kleene based static analysis [LAS00].[6] In this section, we provide a summary of their framework.

The 3-valued-logic framework for program analysis of [SRW02] provides for the automatic generation of abstract interpreters (i.e., analysis algorithms) based on a specification of the programming language's concrete (instrumented) semantics. The most demanding task imposed on the analysis designer is the choice of the memory-state properties that the analysis should track. Essentially, once that choice is made, the rest of the algorithm is synthesized in a provably-correct fashion. In our analyses, we fully utilize the framework of [SRW02] to obtain shape analysis algorithms from a specification of the instrumented concrete semantics and the set of tracked properties. In particular, we utilize their framework to obtain computable conservative abstract transformers.

[6] TVLA is an abbreviation for **T**hree-**V**alued-**L**ogic **A**nalyzer.

In this section, we summarize the framework of [SRW02]. Specificially, we demonstrate how 3-valued logic can serve as the basis for program analysis, by outlining the design of an *intra*procedural shape analysis algorithm for singly-linked list manipulating programs using their framework. The resulting (intraprocedural) shape analysis is an abstract interpretation of the $\mathcal{LSB}$ semantics.[7]

2.5.1 Conservative Representation of Sets of Memory States via 3-Valued Logical Structures

In this section, we explain how an unbounded set of unbounded memory states can be conservatively represented via a bounded set of bounded 3-valued logical structures.

2.5.1.1 Kleene's 3-Valued Logic

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of $\frac{1}{2}$ (unknown) for cases in which predicates could have either value, i.e., 1 (true) or 0 (false). We say that 0 and 1 are *definite* values, whereas $\frac{1}{2}$ is an *indefinite* value. The information partial order on the set $\{0, \frac{1}{2}, 1\}$ is defined as $0 \sqsubseteq \frac{1}{2} \sqsupseteq 1$, and $0 \sqcup 1 = \frac{1}{2}$. Kleene's interpretation of the propositional operators is given in Appendix A.2.

Definition 2.5.1 (Logical Structures) *A 3-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S = \langle U^S, \iota^S \rangle$ where:*

- *U^S is the universe of the 3-valued structure. The universe is comprised of a set of individuals (nodes).*
- *ι^S is an interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p) : (U^S)^k \to \{0, \frac{1}{2}, 1\}$.*

The set of 3-valued *logical structures is denoted by $3Struct$.*

A 2-valued structure is a 3-valued structure with an interpretation limited to $\{0, 1\}$. The set of 2-valued *logical structures is denoted by $2Struct$.*

Unless stated otherwise, we will implicitly assume the use of powerset domains and of set-union as our join operator.[8] Thus, to establish the Galois connection between the powerset domain of program states (ordered by set inclusion) and the powerset domain of $3Struct$, it suffices to show a *representation function* that maps a program state to its "most-precise representation" in $3Struct$.

In this section, we define a representation function $\beta_L^{LSB} : \mathcal{S}_L \to 3Struct$, which maps a memory state of the $\mathcal{LSB}$ semantics to its most precise representation as a 3-valued logical structure, by a composition of two functions:

(i) *to2VLS*$^{LSB} : \mathcal{S}_L \to 2Struct$, which maps a memory state $s_L \in \mathcal{S}_L$ to an unbounded *2-valued* logical structure S, and

(ii) *canonical abstraction* [SRW02]$: 2Struct \to 3Struct$ which conservatively bounds S.

The induced Galois connection $(2^{\mathcal{S}_L}, \alpha : 2^{\mathcal{S}_L} \to 2^{3Struct}, \gamma : 2^{3Struct} \to 2^{\mathcal{S}_L}, 2^{3Struct})$ is defined below, where $\beta_L^{LSB}(s_L) \sqsubseteq S^\sharp$ means that $S^\sharp$ conservatively represents $\beta_L^{LSB}(s_L)$ (see Section 2.5.1.3):

$$\alpha(ss_L) = \{\beta_L^{LSB}(s_L) \mid s_L \in ss_L\} \text{ and } \gamma(SS) = \{s_L \in \mathcal{S}_L \mid S^\sharp \in SS,\ \beta_L^{LSB}(s_L) \sqsubseteq S^\sharp\},$$

2.5.1.2 Representing Memory States via 2-Valued Logical Structures

The function *to2VLS*^{LSB}, defined in Figure 2.10, maps a memory state $s_L = \langle L, \rho, h \rangle \in \mathcal{S}_L$ to a *2-valued* logical structure S. Every location $l \in L$ is represented by a unique individual in U^S. Tracked properties of the memory state are recorded by the *core predicates* (shown in Figure 2.11(a)) and the *instrumentation predicates* (shown in Figure 2.11(b)), which we gradually explain through Example 2.5.2. In this example, we apply *to2VLS*^{LSB} to the memory state depicted in Figure 2.8($s_L^{r\,2.8}$), which may arise at the return site of the first invocation of `splice` in our running example.[9]

[7] The analysis described in this section is an intraprocedural analysis. Thus, it can be similarly casted as an abstract interpretation of the $\mathcal{GSB}$ semantics.

[8] In our implementation, we use a more aggressive join operator than set union. See Sections 3.8.1 and 4.9.1. However, for simplicity, unless stated otherwise, we will implicitly assume the use of set-union as the join operator.

[9] Function *to2VLS*^{LSB}, as defined in Figure 2.10, uses, as an example, predicates which are suitable for representing memory states of procedures manipulating singly linked lists. In general, other predicates could have been used.

$to2VLS^{LSB}: \mathcal{S}_L \rightarrow 2Struct$ s.t.
$to2VLS^{LSB}(\langle L, \rho, h\rangle) = \langle U, \iota\rangle$
where
$U = L$
$\iota : \big((\{ils, c, x, r_x \mid x \in dom(\rho)\} \rightarrow U) \cup (\{n, eq\} \rightarrow U^2)\big) \rightarrow \{0, 1\}$ s.t.

$$
\begin{array}{lcl}
\iota(x)(l) & = & \rho(x) = l \ \text{(for every } x \in dom(\rho)) \\
\iota(n)(l_1, l_2) & = & h(l_1, n) = l_2 \\
\iota(eq)(l_1, l_2) & = & l_1 = l_2 \\
\iota(r_x)(l) & = & \exists l_x \in L,\ \delta \in \Delta \ \text{s.t.}\ \rho(x) = l_x \ \text{and}\ \hat{h}(\langle l_x, \delta\rangle) = l \ \text{(for every } x \in dom(\rho)) \\
\iota(ils)(l) & = & \exists l_1 \in L, l_2 \in L \ \text{s.t.}\ l_1 \neq l_2\ ,\ h(l_1, n) = l \text{ , and } h(l_2, n) = l \\
\iota(c)(l) & = & \exists \delta \in \Delta \ \text{s.t.}\ \delta \neq \epsilon \ \text{and}\ \hat{h}(\langle l, \delta\rangle) = l
\end{array}
$$

Figure 2.10: The function $to2VLS^{LSB}$ maps states in $\mathcal{S}_L$ to 2-valued logical structures using the predicates for list-manipulating programs shown in Figure 2.11. (See Definition 2.4.3 for the definition of function $(\hat{h})$).

Predicate	Intended Meaning
$x(v)$	The reference variable x points to the object v
$n(v_1, v_2)$	The n-field of object v_1 points to object v_2
$eq(v_1, v_2)$	v_1 and v_2 are the same object

(a) Core predicates.

Predicate	Intended Meaning
$r_x(v)$	v is reachable from variable x
$ils(v)$	v is *locally* shared, i.e., v is pointed-to by a field of more than one object in the *local-heap*
$c(v)$	v resides on a directed cycle of fields

(b) Instrumentation predicates.

Figure 2.11: The predicates used in the analysis of list-manipulating programs. (a) The core predicates. There is a separate predicate x for every local variable x used by the program. Note that predicate n is the only one that is specific to the linked list data structure. The remaining predicates would play a rule in the analysis of any data structure. When there is no risk of confusion, we sometimes use in our formulae the more intuitive infix notations $v_1 = v_2$ and $v_1 \neq v_2$ instead of $eq(v_1, v_2)$ and $\neg eq(v_1, v_2)$, respectively. (b) The instrumentation predicates. There is a separate predicate r_x for every local variable x used by the program.

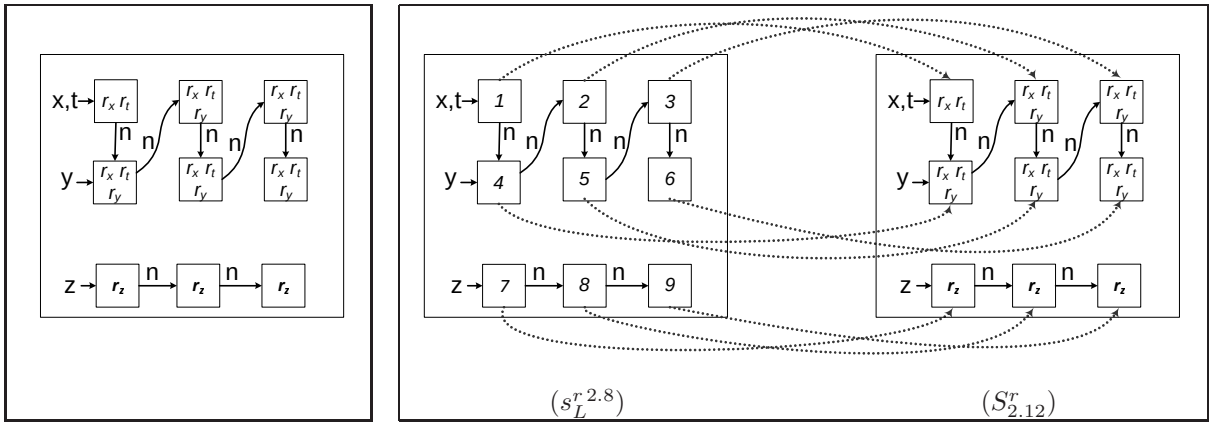


Figure 2.12: $(S^r_{2.12})$: The *2-valued* logical structure $S^r_{2.12} = \langle U^{S^r_{2.12}}, \iota^{S^r_{2.12}} \rangle = to2VLS^{LSB}(s_L^{r\,2.8})$ conservatively represents the memory state $s_L^{r\,2.8}$ shown in Figure 2.8. (Note that $S^r_{2.12}$ does not track the values of *locations* of objects). $(to2VLS^{LSB}\colon L^{s_L^{r\,2.8}} \to U^{S^r_{2.12}})$: The mapping $to2VLS^{LSB}\colon L^{s_L^{r\,2.8}} \to U^{S^r_{2.12}}$ is depicted by the dotted arrows going from the locations of $s_L^{r\,2.8}$ to the individuals of $S^r_{2.12}$: The value of $to2VLS^{LSB}$ at location $l \in L^{s_L^{r\,2.8}}$ is depicted as a dotted arrow emanating from l and pointing to the node $u \in U^{S^r_{2.12}}$.

Example 2.5.2 Figure 2.12($S^r_{2.12}$) depicts the resulting *2-valued* logical structure when $to2VLS^{LSB}$ is applied to the memory state $s_L^{r\,2.8}$ depicted in Figure 2.8($s_L^{r\,2.8}$), i.e., $S^r_{2.12} = to2VLS^{LSB}(s_L^{r\,2.8})$. Figure 2.12($to2VLS^{LSB}\colon L^{s_L^{r\,2.8}} \to U^{S^r_{2.12}}$) depicts the mapping induced by $to2VLS^{LSB}$.

A *2-valued* logical structure $S = \langle \iota^S, U^S \rangle$ is depicted as a directed graph. A directed edge between nodes u_1 and u_2 that is labeled with binary predicate symbol p indicates that $\iota^S(p)(u_1, u_2) = 1$. Also, for a unary predicate symbol p, we draw p inside a node u when $\iota^S(p)(u) = 1$; conversely, when $\iota^S(p)(u) = 0$ we do not draw p in u. The universe of the 2-valued logical structure $S^r_{2.12}$ contains nine individuals representing the nine list nodes.

Core Predicates

Tracked properties of the memory state are recorded by the *core predicates* given in Figure 2.11(a). The core predicates are x, t, y, z, n, and eq:

- For each pointer variable x, there is a unary predicate x. The value of $\iota^{S^r_{2.12}}(x)(u)$ is 1 if variable x points-to the list element represented by u in $U^{S^r_{2.12}}$. The value of the x-predicate is depicted via an edge from the predicate name x to the node that represents the list element that x points-to.
- The pointed-to-by-a-field relation between list elements is represented by the binary predicate n, i.e., $\iota^{S^r_{2.12}}(n)(u_1, u_2) = 1$ if the n-field of the list element represented by u_1 points-to the list element represented by u_2. Note that predicate n is the only one that is specific to the linked list data structure. In general, we would have a binary predicate f for every field f defined in the program.
- The binary predicate eq records the equality relation. We note that because $S^r_{2.12}$ is a 2-valued logical structure, then $\iota^{S^r_{2.12}}(eq)(u_1, u_2) = 1$ if and only if $u_1 = u_2$. The main role of this predicate is to simplify the definition of the bounded abstraction, as is shortly explained. The values of the predicate eq are not depicted in the graphical depiction of *2-valued* logical structures.

Instrumentation Predicates

The instrumentation predicates are r_x, r_q, and ils, and c. They are an adaptation to local-heaps of the standard predicates used in the analysis of singly linked lists [LARSW00, LAS00, SRW02]. The predicates are shown in Figure 2.11(b).

The *instrumentation principle* [SRW02, Observation 2.8] states that it is sometimes advantageous to explicitly "store" in a *2-valued* logical structure predicates that record information which can be derived from the values of the core predicates. Intuitively, the reason is that *instrumenting* S with such derived information can make its *canonical abstraction* more precise, as explained in Section 2.5.1.3.

- The unary predicate r_x holds for list elements that are reachable by an access path that starts at a local variable x of the *current* call. In $s_L^{r\,2.8}$, variables x and t point to the (same) list containing 6 elements. Thus, in $S^r_{2.12}$, the value of the predicates r_x and r_t are 1 for all the nodes that represent the elements of this list. Variable y points to the second element in this list. Thus, in $S^r_{2.12}$, the value of the predicate r_y is 0 for the individual representing the head of that list.
- The unary predicate ils captures *local-heap* sharing information. The predicate has the value 1 at a node u that represents a list element that is pointed-to by the n-fields of two or more list elements in the *local-heap*. In $s_L^{r\,2.8}$, no list element is locally shared. Thus, the value of $\iota^{S^r_{2.12}}(ils)$ is 0 for all of the individuals in $U^{S^r_{2.12}}$. (In this example, the list nodes are also not globally shared. In Section 4.7 we give an example where there is a node which is globally shared, but is not locally shared.)
- The unary predicate c holds at an individual that resides on a cycle of n-fields. Because both lists in $s_L^{r\,2.8}$ are acyclic, $\iota^{S^r_{2.12}}(c)$ is 0 for all the individuals.

2.5.1.3 Representing a Local-Heap by a 3-Valued Logical Structure using Canonical Abstraction

The main idea in canonical abstraction is to represent several list elements by a single node, i.e., the mapping from list elements to the universe of the *3-valued* logical structure is a surjective, but not necessarily an injective, function. A node that may represent more than one list element is called a *summary* node.

Definition 2.5.3 (Embedding [SRW02]) *A* 3-valued *logical structure* $S^\sharp = \langle \iota^{S^\sharp}, U^{S^\sharp} \rangle$ *is* ***embedded*** *into a* 3-valued *logical structure* $S'^\sharp = \langle \iota^{S'^\sharp}, U^{S'^\sharp} \rangle$*, denoted by* $S^\sharp \sqsubseteq S'^\sharp$*, if there exists a surjective function* $f\colon U^{S^\sharp} \to U^{S'^\sharp}$ *such that for all predicates* $p \in \mathcal{P}$ *of arity* k *and for all* k*-tuples* $u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp \in U^{S^\sharp}$

$$\iota^{S'^\sharp}(p)(u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp) = \iota^{S^\sharp}(p)(f(u_1^\sharp), f(u_2^\sharp), \ldots, f(u_k^\sharp)) \text{ or } \iota^{S'^\sharp}(p)(f(u_1^\sharp), f(u_2^\sharp), \ldots, f(u_k^\sharp)) = 1/2.$$

We say that $S'^\sharp$ ***conservatively represents*** $S'^\sharp$ *and that a node* $u^\sharp \in U^{S^\sharp}$ ***represents*** *node* $u \in U$ *when* $f(u) = u^\sharp$.

Informally, the *3-valued* logical structure $S^\sharp$ is the best conservative representation a memory-state s_L under canonical abstraction is obtained by "merging" *all* the nodes in the *2-valued* logical structure $S = \mathit{to2VLS}^{LSB}(s_L)$ that have the same values for *all* the unary predicates (and using these values for the unary predicates at the "merged" node). The value of a binary predicate $\iota^{S^\sharp}(p)(u_1^\sharp, u_2^\sharp)$ is set to a *definite* value (0 or 1) only when the predicate $\iota^S(p)(u_1, u_2)$ has this value for all the nodes u_1 and u_2 in U^S that are "merged" into $u_1^\sharp$ and $u_2^\sharp$, respectively.

Definition 2.5.4 (Canonical Abstraction [SRW02]) *A* 3-valued *logical structure* $S^\sharp$ *is a* ***canonical abstraction*** *of a* 2-valued *logical structure* S *if there exists a surjective function* $f\colon U^S \to U^{S^\sharp}$ *satisfying the following conditions:*

(i) for all $u_1, u_2 \in U^S$, $f(u_1) = f(u_2)$ *iff for all unary predicates* $p \in \mathcal{P}$, $\iota^S(p)(u_1) = \iota^S(p)(u_2)$, *and*

(ii) for all predicates $p \in \mathcal{P}$ *of arity* k *and for all* k*-tuples* $u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp \in U^{S^\sharp}$,

$$\iota^{S^\sharp}(p)(u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp) = \bigsqcup_{\substack{u_1,\ldots,u_k \in U^S \\ f(u_i)=u_i^\sharp}} \iota^S(p)(u_1, u_2, \ldots, u_k).$$

Note that by definition, every *2-valued* logical structure has a *3-valued* logical structure that is its canonical abstraction.

Example 2.5.5 The *3-valued* logical structure $S^{r\sharp}_{2.13}$, depicted in Figure 2.13, is a canonical abstraction of the *2-valued* logical structure $S^r_{2.12}$.

3-valued logical structures are also drawn as directed graphs. Definite values are drawn as for 2-valued structures. Binary indefinite predicate values ($\frac{1}{2}$) are drawn as dotted directed edges. Summary nodes are depicted by a double frame.

The universe of $S^r_{2.12}$ contains nine individuals. The individuals at the tail of the list pointed to by y resp. by z have the same values for all the unary predicates. Thus, the universe of $S^{r\sharp}_{2.13}$ contains five nodes. Figure 2.13 also depicts the mapping $f\colon U^{S^r_{2.12}} \to U^{S^{r\sharp}_{2.13}}$ induced by the canonical abstraction of $S^r_{2.12}$ by $S^{r\sharp}_{2.13}$. Note that the value of every unary predicate is the same for a node $u \in U^{S^r_{2.12}}$ and for the node that represents u in $U^{S^{r\sharp}_{2.13}}$.

$S^{r\sharp}_{2.13}$ has two summary nodes: The summary node at the top part of the diagram represents all the elements in the tail of the list pointed to by y. The other summary node represents all the elements in the tail of the list pointed to by z. The fact that a node is a summary node is recorded by the predicate eq, which has an indefinite value at a summary node, i.e., $\iota^{S^{r\sharp}_{2.13}}(eq)(u,u) = 1/2$. The value of the n-field emanating from the list element pointed to by y and pointing to the summary node is indefinite because the n-field of the list element pointed to by y points-to its immediate successor in the list, but not to the other list elements in the tail, which are also represented by that same summary node. The value of the n-field emanating from the summary node and also pointing to it is indefinite for similar reasons.

The chosen predicates serve as examples to show that the abstraction maintains some sorts of information and loses other. For example, we can see that in any memory state represented by $S^{r\sharp}_{2.13}$ there is no garbage (e.g., all the list elements are reachable from either x or z, as indicated by the fact that in every individual either predicate r_x holds or predicate r_z); the list pointed-to by x (resp. z) is acyclic (predicate c does not hold in any node); and that x and z point to disjoint lists (r_x and r_z do not hold together in any list element). However, we no longer know the number of elements in a list.

2.5.2 Abstract Interpretation of Program Statements

In this section, we explain how logical formulae can be used to extract information pertaining to a program memory state from a 3-valued logical structure that conservatively represents this state. We then show how the meaning of program statements as transformers from logical structures to logical structures can be defined by a collection of first order formulae with transitive closure.

2.5.2.1 Expressing Properties via Formulae

Properties of structures can be extracted by evaluating formulae. We use first-order logic with transitive closure and equality, but without function symbols and constant symbols. The formal definition for the syntax of formulae and the definition for Kleene's 3-valued logic are shown in Appendix A.2.

For example, the formula

$$\exists v_1, v_2 : \quad \neg eq(v_1, v_2) \wedge n(v_1, v) \wedge n(v_2, v) \tag{2.1}$$

expresses the fact that list element v is pointed to by more than one n-field.

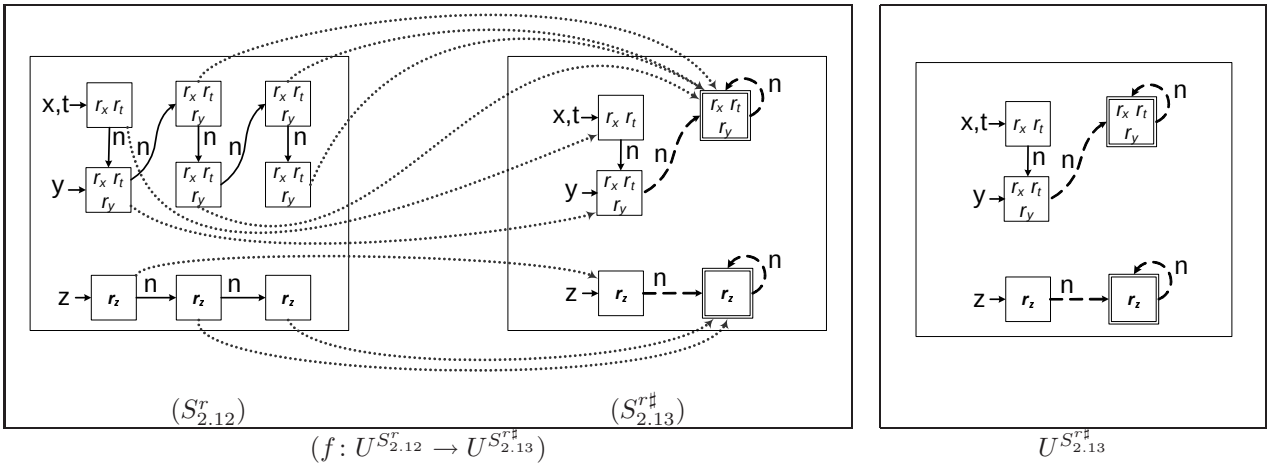


Figure 2.13: The *2-valued* logical structure $S^r_{2.12} = \langle U^{S^r_{2.12}}, \iota^{S^r_{2.12}} \rangle = \mathit{to2VLS}^{LSB}(s_L^{r\,2.8})$ conservatively represents the memory state $s_L^{r\,2.8}$ shown in Figure 2.8. (Note that $S^r_{2.12}$ does not track the values of *locations* of objects). The *3-valued* logical structure $S^{r\sharp}_{2.13} = \langle U^{S^{r\sharp}_{2.13}}, \iota^{S^{r\sharp}_{2.13}} \rangle$ is the canonical abstraction of $S^r_{2.12}$. The mapping $f\colon U^{S^r_{2.12}} \to U^{S^{r\sharp}_{2.13}}$ induced by the canonical abstraction of $S^r_{2.12}$ by $S^{r\sharp}_{2.13}$ is depicted by the dotted arrows going from $S^r_{2.12}$ to $S^{r\sharp}_{2.13}$: The value of f at node $u \in U^{S^r_{2.12}}$ is depicted as a dotted arrow emanating from $u \in U^{S^r_{2.12}}$ and pointing to the node $f(u) \in U^{S^{r\sharp}_{2.13}}$ which represents u in $S^{r\sharp}_{2.13}$.

Predicate	**Defining Formula**
$r_x(v)$	$\exists v_x\colon x(v_x) \wedge n^*(v_x, v)$
$\mathit{ils}(v)$	$\exists v_1, v_2\colon \neg \mathit{eq}(v_1, v_2) \wedge n(v_1, v) \wedge n(v_2, v)$
$c(v)$	$\exists v_1\colon n(v, v_1) \wedge n^*(v_1, v)$

Figure 2.14: The defining formulae for the instrumentation predicates used in the analysis of list-manipulating programs. The intended meaning of the predicates is described in Figure 2.11(b). n^* is a shorthand for the reflexive transitive closure of the binary predicate n. (See Section A.2).

The Embedding Theorem (see [SRW02, Theorem 4.9]) states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all of the 2-valued structures which are conservatively represented by that structure. The Embedding Theorem is the foundation for the use of 3-valued logic in static-analysis: It ensures that it is sensible to take a formula that—when interpreted in 2-valued logic—defines a property, and reinterpret it on a 3-valued structure S: The Embedding Theorem ensures that one must obtain a value that is conservative with regard to the value of the formula in any 2-valued structure represented by S.

Example 2.5.6 Consider the 2-valued structure $S^r_{2.12}$ shown in Figure 2.13. The formula (2.1) evaluates to 0 at all of the list nodes.

In contrast, consider the 3-valued structure $S^{r\sharp}_{2.13}$ shown in Figure 2.13. Formula (2.1) evaluates to $1/2$ at the summary nodes. This is in line with the Embedding Theorem since $1/2$ is an indefinite value while 0 is a definite value. However, it is not very precise since the it loses the fact that no list element is shared.

Note, however, that by explicitly recording the values of the instrumentation predicates in the *2-valued* logical structure $S^r_{2.12}$, its canonical abstraction still maintains the information that no list node is locally shared. This is one example for the way storing derived information in the concrete structure helps improve the precision of its canonical abstraction. Formally, every instrumentation predicate is associated with a formula in first-order logic with transitive closure which defines its intended meaning. Figure 2.14 provides the defining formulae for the instrumentation predicates shown in Figure 2.11(b).

$$\langle st, SS\rangle \overset{3VL^\sharp}{\rightsquigarrow} \{\beta_L(s'_L) \mid s_L \in \gamma(SS), \langle st, s_L\rangle \overset{LSB}{\rightsquigarrow} s'_L\}$$

Figure 2.15: A specification of the abstract inference rules for atomic statements.

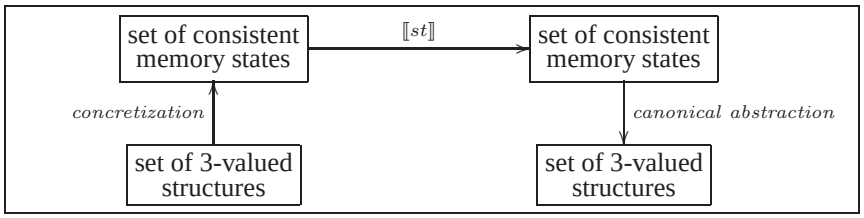


Figure 2.16: The best abstract semantics of a statement `st` with respect to canonical abstraction. $[\![st]\!]$ is the operational semantics of `st` applied pointwise to every consistent memory state.

2.5.2.2 The Meaning of Program Statements

The meaning-functions for program statements are defined as transformers from 2-valued structures to 2-valued structures. Properties of memory states can be obtained by evaluating first order logical formulae against the representing structure (see Section A.2), thus, these transformers are defined by a collection of first order formulae evaluated against the original structure. The value of every predicate is determined by a corresponding formula. The main idea is that if a structure $S^\sharp$ represents a set of memory states that arise before statement st, then a structure $S'^\sharp$ that represents the corresponding set of memory states that arise after st can be obtained by evaluating a suitable collection of formulae that capture the semantics of st. We defer the explanation of the use of logic to define the operational semantics of procedure calls to Chapter 3.

Consistent 2-Valued Structures. Some 2-valued structures cannot represent memory states, e.g., when a unary predicate x holds at two different nodes for a local variable `x`. A 2-valued structure is *consistent* if it can represent a memory state. It turns out that the analysis can be more precise by eliminating inconsistent 2-valued structures.

Declarative Definition of the Abstract Transformers. The specification of the abstract interpretation is given by "abstract" inference rules in the same style as the natural semantics. The abstract inference rules operate on *3-valued* logical structures. Figure 2.15 shows the specification of the abstract inference rules for atomic statements. These rules are declarative in the style of the best abstract transformer [CC79], where every abstract inference rule emulates a corresponding concrete inference rule using represented states.

Conceptually, the most precise (also called *best* or *induced*) conservative effect of a program statement on a set of 3-valued logical structures SS is defined in three stages, depicted in Figure 2.16:

(i) find each consistent memory state s_L represented by a 3-valued logical structure $S \in SS$ (*concretization*);
(ii) apply the statement's concrete operational semantics to every such state s_L, and
(iii) abstract each of the resulting memory states by a 3-valued structure (*canonical abstraction*).

2.5.2.3 Implementation

Our analysis algorithms do not explicitly apply the concrete operational semantics to each of the (potentially infinite number of unbounded) structures represented by a three-valued structure S. Instead, it applies it to the bounded *3-valued* logical structures that arise during the anlaysis. In addition, it uses a partial concretization [LAIS06, LASIR07] operation (*Focus*) and a semantic reduction operation based on the notion of consistent 2-valued logical structures (*Coerce*). These operations are part of the 3-valued logical framework of [SRW02] and its implementation in the TVLA system [LAS00].

Chapter 3

Interprocedural Local-Heap Shape Analysis for Cutpoint-Free Programs

This chapter presents a framework for interprocedural shape analysis, which is context- and flow-sensitive with the ability to perform destructive pointer updates. In this chapter, we limit our attention to cutpoint-free programs—programs in which in every procedure invocation, the objects pointed to by the actual parameters dominate the procedure's local-heap (i.e., the part of the heap reachable from the actual parameters).

Technically, our analysis computes procedure summaries as transformers from inputs to outputs while ignoring parts of the heap not relevant to the procedure. *This makes the analysis modular in the heap and thus allows reusing the effect of a procedure at different call-sites and even between different contexts occurring at the same call-site. We note that our analysis also verifies that a program is cutpoint-free. This makes the analysis applicable for arbitrary programs (i.e., it does not require an a priori classification of a program as cutpoint-free).*

The material described in this chapter is largely based on the material that originally appeared in [RSY05a, RSY05b].

3.1 Introduction

In this chapter, we introduce a new approach for shape analysis for a class of imperative programs. The main idea is to restrict the "sharing patterns" occurring in procedure calls between the procedure's local-heap and the irrelevant context to include only the objects passed as parameters. This restriction, which we refer to as *cutpoint-freedom*, simplifies the verification problem by allowing procedures to be analyzed ignoring the parts of the heap not reachable from actual parameters. Moreover, shape analysis can conservatively detect violations of the above restrictions, thus allowing to treat existing programs.

Technically, in this chapter we present $\mathcal{LSL}^{\text{CPF}}$, a non-standard concrete storeless semantics, which *checks* that no procedure invocation in an execution yields a cutpoint. We then develop a framework for interprocedural shape analysis by abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$.

We describe the algorithm in the context of the analysis of programs that manipulate singly linked lists.[1] The algorithm finds a finite description of all the memory states that arise during program execution. Useful information regarding the program's behavior can be extracted from the computed descriptors. In addition, they can be used to conservatively verify that a program is cutpoint-free. Consider, for example, the program used as the running example of Chapter 2 (see Figure 2.2), and which is used as a running example in this chapter too. Some of the properties that our analysis of this program successfully determines are that it does not dereference null-valued pointers; does not create garbage; and that when, e.g., the first call to `splice` returns, the variables `x` and `t` point-to an acyclic linked list whose second element is pointed to by `y`. In addition, the algorithm verifies that this program is cutpoint-free, while its variant, shown in Figure 3.1(b), may not be.

[1] We note that we have applied our algorithm to analyze tree-manipulating programs and sorting programs. See Section 3.8.1.

The analysis benefits from the fact that in $\mathcal{LSL}^{\text{CPF}}$ the heap is localized, i.e., the behavior of a procedure only depends on the contents of its local-heap. This allows analysis results to be reused for different contexts and makes the analysis more likely to scale up.

Our algorithm is parametric in the heap abstraction and in the concrete effects of program statements, allowing to experiment with different instances of interprocedural shape analyzers. For example, we can employ different abstractions for singly-, doubly-linked lists, and trees.

The algorithm is presented in terms of the 3-valued-logic framework for program analysis of [SRW02]. This framework automatically generates abstract interpreters (i.e., analysis algorithms) based on a specification of the programming language's concrete (instrumented) semantics. Furthermore, the combination of the theorems given in [SRW02] with our results guarantee the soundness of every instance of our analysis (see Section 3.7.3).

The most demanding task on the analysis designer is the choice of the memory-state properties that the analysis should track. Once the choice is made, the rest of the algorithm is synthesized in a provably-correct fashion. Thus, we do not give the full details of the analyses. Instead, in Section 3.6, we focus on the *canonical abstraction* [SRW02] (see Section 2.5.1.3) of $\mathcal{LSL}^{\text{CPF}}$ memory states; in Section 3.7, we describe the abstract semantics; and in Section 3.8, we describe the tabulation-based implementation of our static analyzer.

This chapter consists of two main parts: The first part defines the notion of *cutpoint-freedom* and introduces a non-standard concrete storeless semantics, $\mathcal{LSL}^{\text{CPF}}$, for *Localized-heap Store-Less CutPoint-Free*. In $\mathcal{LSL}^{\text{CPF}}$, called procedures are only passed *parts* of the heap. $\mathcal{LSL}^{\text{CPF}}$ *forbids cutpoints*: In every procedure invocation, it checks whether the invocation yields a cutpoint, and, if so, it aborts the execution. The second part concerns abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$. It develops new static-analysis algorithms using canonical abstraction [SRW02] and uses these algorithms to verify interesting properties of cutpoint-free programs, including partial correctness of a recursive quicksort implementation.

3.1.1 Cutpoint-Freedom

We introduce an analysis method in which procedures operate on local-heaps containing only the objects reachable from actual parameters. One of the most complex aspects of local-heap shape analysis is the treatment of sharing between the local-heap and the rest of the heap. The problem is that the local-heap can be accessed via access paths which bypass actual parameters. Therefore, special care need to be given to *cutpoints*, objects in the local-heap which separate the local-heap that can be accessed by a procedure from the rest of the heap which—from the viewpoint of that procedure—is non-accessible and immutable. (See Definition 3.3.2). In this chapter, we simplify the analysis problem by forbidding cutpoints: We develop shape analyses which are targeted at a restricted class of programs which never have cutpoints.

We refer to a procedure invocation which does not yield a cutpoint object as a *cutpoint-free invocation*. We refer to an execution of a program in which all invocations are cutpoint-free as a *cutpoint-free execution*, and to a program in which all executions are cutpoint-free as a *cutpoint-free program*.

While many programs are not cutpoint-free, we observe that a reasonable number of programs, including all examples used in [DRS00, RS01, JLRS04] are cutpoint-free, as well as many of the programs in [Deu94, SYKS03]. One of the key observations in this chapter, is that we can benefit from cutpoint-freedom to construct an interprocedural shape analysis algorithm that efficiently reuses procedure summaries.

Technically, in this section we present $\mathcal{LSL}^{\text{CPF}}$, a non-standard concrete *storeless* operational semantics that efficiently handles cutpoint-free programs. This semantics is interesting because procedures operate on local-heaps, thus supporting the notion of heap-modularity while permitting the usage of a global-heap and destructive updates.[2] $\mathcal{LSL}^{\text{CPF}}$ checks that a program execution is indeed cutpoint-free and halts otherwise. As a result, it is applicable to any arbitrary program, and does not require an a priori classification of a program as cutpoint-free. We show that for cutpoint-free programs, $\mathcal{LSL}^{\text{CPF}}$ is observationally equivalent to the standard global-heap semantics.

3.1.2 $\mathcal{LSL}^{\text{CPF}}$: A Localized-Heap Storeless Cutpoint-Free Semantics

In this chapter, we present a first step towards a storeless semantics that allows representation of parts of the heap *and* relating properties before and after a call. Our storeless semantics addresses the problem of relating properties

[2] As we shall see in Chapter 4, the absence of cutpoints drastically simplifies the meaning of procedure calls.

of memory cells before and after a call by placing certain restrictions on the allowed aliasing that may occur at procedure calls. More specifically, every procedure call is checked to be cutpoint-free.[3] This ensures that for computing the effect of the procedure, it suffices to only keep track of the objects which, at the entry state, are pointed-to by a formal parameter.

3.1.3 Interprocedural Shape Analysis for Cutpoint-Free Programs

In this chapter, we present a framework for interprocedural shape analysis, which is context- and flow-sensitive with the ability to perform destructive pointer updates. Our analysis computes procedure summaries as transformers from input local-heaps to output local-heaps, thus abstracting away parts of the heap not relevant to the procedure. This makes the analysis modular in the heap (*heap modular*) and thus allows reusing the effect of a procedure at different call-sites and even between different contexts occurring at the same call-site.

The absence of cutpoints drastically simplifies the meaning of procedure calls (compare, e.g., the call rule in Section 3.4.2.3 and in Section 4.4.2.3). A beneficial byproduct is that it also simplifies the task of designing an interprocedural shape analysis algorithm. Indeed, $\mathcal{LSL}^{\text{CPF}}$ gives rise to a simple interprocedural shape-analysis for cutpoint-free programs.

Furthermore, restricting our attention to cutpoint-free programs reduces the asymptotic complexity of the interprocedural shape analysis: For programs without global variables, the worst-case time complexity of the analysis is doubly exponential in the maximum number of local variables in a procedure, instead of being doubly exponential in the overall number of local variables [RS01].

Technically, our algorithm is built on top of the 3-valued logical framework for program analysis of [LAS00, SRW02]. Thus, it is parametric in the heap abstraction and in the concrete effects of program statements, allowing to experiment with different instances of interprocedural shape analyzers. For example, we can employ different abstractions for singly-, doubly-linked lists, and trees. The soundness of our approach is immediate from the soundness of the more general case of programs with cutpoints, presented in Chapter 4 (see Sections 3.5 and 3.7).

We implemented a prototype of our analysis and used to verify properties that could not be automatically verified before. We provide an initial empirical evaluation of our algorithm. Our empirical evaluation indicates that the analysis is precise enough to prove properties such as the absence of null dereferences, preservation of data structure invariants such as list-ness, tree-ness, and sorted-ness for iterative and recursive programs with deep references into the heap and destructive updates (including the partial correctness of a recursive quicksort [Hoa61] implementation, i.e., show that it returns an ordered permutation of its input). We observe that (in our experiments) the cost of analyzing recursive procedures is comparable to the cost of analyzing their iterative counterparts. Moreover, the cost of analyzing a program with procedures is smaller than the cost of analyzing the same program with procedure bodies inlined.

3.1.4 Main Results

The main results described in this chapter can be summarized as follows:

- We define the notion of cutpoint-free programs, and show that interesting cutpoint-free programs can be written naturally, e.g., all programs verified using shape analysis in [DRS00, RS01, JLRS04], and many of those in [Deu94, SYKS03].
- We develop a non-standard *storeless concrete* semantics, $\mathcal{LSL}^{\text{CPF}}$, for *Cutpoint-Free Localized-heap Store-Less*. $\mathcal{LSL}^{\text{CPF}}$ is a *procedure local-heap semantics*: a procedure is not passed parts of the heap which it cannot reach. $\mathcal{LSL}^{\text{CPF}}$ also checks that every procedure invocation is cutpoint-free.
- We present an interprocedural shape analysis for cutpoint-free programs by abstract interpretation of $\mathcal{LSL}^{\text{CPF}}$.
- We empirically evaluated the analysis by implementing a prototype and experimenting with several small—yet intricate—Java programs, including programs manipulating unshared trees and a recursive implementation of quicksort. Preliminary experimental results indicate that: (i) the cost of analyzing recursive procedures is similar to the cost of analyzing their iterative versions; (ii) our analysis benefits from procedural abstraction; (iii) our approach compares favorably with [RS01, JLRS04].

[3] Note that in Chapter 4, we lift this restriction and develop a (more complicated) procedure local-heap storeless semantics for programs with a (possibly unbounded) number of cutpoints.

```
public class MainCutpointFree{
   public static void main(String[] argv) {
      List x = create3(1);
      List y = create3(4);
      List z = create3(7);
      List t = splice(x, y);
      List s = splice(y, z);
   }
}
```

(a) A cutpoint-free Java program.

```
public class MainNotCutpointFree{
   public static void main(String[] argv) {
      List x = create3(1);
      List y = create3(4);
      List z = create3(7);
      List t = splice(x, y);
      List s = splice(t, z);
   }
}
```

(b) A non cutpoint-free Java program.

Figure 3.1: Two JAVA programs that create and splice three singly-linked lists: (a) a cutpoint-free program. (b) a program with cutpoints. The statement which differ the two programs is written in bold. The `List` class is shown in Figure 2.2(b).

Outline. The remainder of the chapter is organized as follows: Section 3.2 introduces our running example. Section 3.3 discusses the notions of cutpoints and of cutpoint-freedom. Section 3.4 defines the $\mathcal{LSL}^{\text{CPF}}$ semantics and investigates its properties. Sections 3.6, 3.7, and 3.8 present our interprocedural shape analysis.

3.2 Motivating Example

Figure 3.1 shows two JAVA programs that splice three unshared, disjoint, acyclic singly-linked lists using a recursive `splice` procedure. (The code of the `List` class is shown in Figure 2.2). Figure 3.1(a) is, essentially, the same program used as the running example in Chapter 2 (see Section 2.1.1). This program is cutpoint-free. We use this program as the running example in this chapter too. Figure 3.1(b) is a variant of our running example which is not cutpoint-free.

Our analyzer verifies that the program shown in Figure 3.1(a) is cutpoint-free. It also detects that its variant, shown in Figure 3.1(b), may not be cutpoint-free.

For each invocation of `splice` in the running example, our analyzer verifies that the returned list is acyclic and not heap-shared;[4] that the first parameter is aliased with the returned reference; and that the second parameter points to the second element in the returned list.

For this example, our algorithm effectively reuses procedure summaries, and only analyzes `splice(p,q)` once for every possible abstract input. As shown in Section 3.8, this means that `splice(p, q)` will be only analyzed a total number of 9 times. This should be contrasted with [RS01], in which no summaries are computed, and the procedure is analyzed 66 times. Compared to [JLRS04], our algorithm can summarize procedures in a more compact way (see Section 6.3).

[4] An object is heap-shared if it is pointed-to by a field of more than one object.

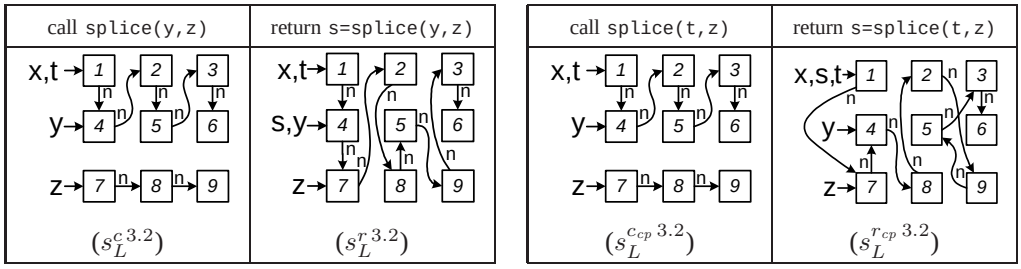


Figure 3.2: Concrete states for: (a) the invocation `s = splice(y, z)` in the program of Figure 3.1(a); (b) the invocation `s = splice(t, z)` in variant of our running example, shown at Figure 3.1(b).

3.3 Cutpoints and Cutpoint-Freedom

In this section, we define cutpoints and discuss the advantages of cutpoint-freedom for a storeless semantics. To assist the reader, we provide some intuition by referring to the global-heap and local-heap store-based semantics (see Sections 2.2 and 2.3, respectively) and to a small-step stack-based operational semantics.

3.3.1 Local-heaps, Relevant Objects, Cutpoints, and Cutpoint-freedom

In our semantics, procedures operate on local-heaps. The local-heap contains only the part of the program's heap accessible to the procedure. Thus, procedures are invoked on local-heaps containing only objects reachable from actual parameters. We refer to these objects as the *relevant* objects for the invocation.

> **Example 3.3.1** Figure 2.8 shows the concrete memory states that occur at the call `t = splice(x, y)` according to the $\mathcal{LSB}$ semantics (see Section 2.3). Figure 2.8($s_L^{c\,2.8}$) shows the state at the point of the call, and Figure 2.8($s_L^{e\,2.8}$) shows the state on entry to `splice`. Here, `splice` is invoked on local-heap containing the (relevant) objects reachable either from `x` or from `y`.

The fact that the local-heap of the invocation `t = splice(x, y)` is separated from the rest of the heap by the objects pointed to by `x` and `y` guarantees that destructive updates performed by `splice` can only affect access paths that pass through an object pointed to by either `x` or `y`. Similarly, the invocation `s = splice(y, z)` in the concrete memory state $s_L^{c\,3.2}$, shown in Figure 3.2(a), can only affect access paths that pass through an object pointed to by either `y` or `z`.

Obviously, this is not always the case. For example, consider a variant of the example program, shown in Figure 3.1(b), in which the second call `s = splice(y, z)` is replaced by a call `s = splice(t, z)`. Figure 3.2($s_L^{r_{cp}\,3.2}$) depicts the concrete memory state which arises when `s = splice(t, z)` is invoked on the concrete memory state shown in Figure 3.2($s_L^{c_{cp}\,3.2}$).

As shown in the figures, the destructive updates of the `splice` procedure change not only paths from `t` and `z`, but also change the (pending) access paths from `y` (of length 1 or more). This indirect change happens because the object pointed-to by `y` at the call `s = splice(t, z)` (Figure 3.2(b)) is a *cutpoint*.

Definition 3.3.2 (Cutpoints) *A **cutpoint** for an invocation of procedure p is a heap-allocated object that, in the program state in which the execution of p's body starts, is: (i) reachable from a formal parameter of p (but not pointed-to by one) and (ii) pointed-to by a pending access path that does not* pass through *any object that is reachable from one of p's formal parameters.*

In other words, a cutpoint is a relevant object that separates the part of the heap which is passed to the callee from the rest of the heap, but which is not pointed-to by a parameter.

For example, the list element pointed to by `y` is a cutpoint when `s = splice(t, z)` is invoked in memory state $s_L^{c_{cp}\,3.2}$ because it is pointed-to by `y`, a local variable of the caller, but it is not pointed to by either one of the actual parameters, `t` or `z`. Thus, this invocation is not *cutpoint-free*. In contrast, the call `t = splice(x, y)` in

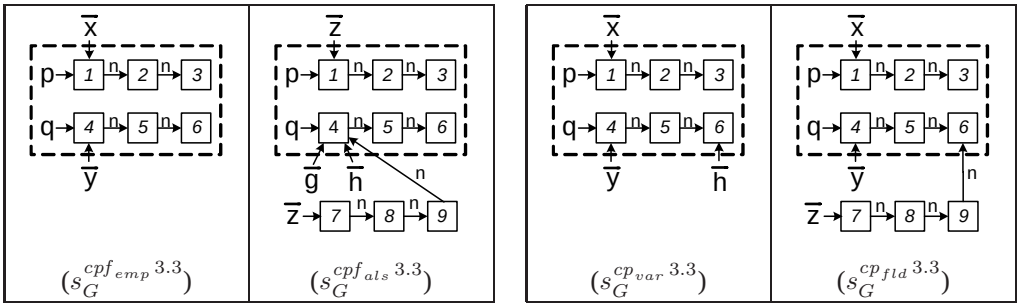


Figure 3.3: Potential *global-heaps* that can (possible global-heaps) and cannot (impossible global-heaps) be represented by the concrete local-heap $s_L^{e\,2.8}$, shown at Figure 2.8. The local-heap of $s_L^{e\,2.8}$ is depicted in the global-heap circumscribed with a dashed framed. Also, for clarity, we use the notation $\overline{\mathsf{x}}$ to denote a reference variable x of a pending call.

memory state $s_L^{c\,3.2}$, shown in Figure 3.2, does not have any cutpoints and is therefore *cutpoint-free*. In fact, all invocations in the program shown in Figure 3.1(a), including recursive ones, are cutpoint-free, and the program is a cutpoint-free program.

Our analyzer verifies that the running example is a cutpoint-free program. It also detects that in the variant of our running example, shown in Figure 3.1(b), the call `s = splice(t, z)` is not a cutpoint-free invocation.[5]

3.3.2 A Global View of Cutpoint-Free Local-Heaps

The key reason for the soundness of our approach is that every local-heap represents all the *global* memory configurations containing that local-heap. Figure 3.3 illustrates that for the local-heap $s_L^{e\,2.8}$, shown at Figure 2.8. We give two examples of potential global memory states represented by this local-heap and two examples which are not represented by this local-heap. Note that here, we actually draw the global-heap, i.e., we draw all the allocated objects.

The two left memory states are represented by the local-heap $s_L^{e\,2.8}$. Memory state $s_G^{cpf_{emp}\,3.3}$ occurs when there are no irrelevant objects, i.e., `splice`'s local-heap contains all the allocated objects. Memory state $s_G^{cpf_{als}\,3.3}$ is also possible. Note that the relevant object 4 is pointed to by two variables, and by the n-field of object 9. This sharing is allowed, i.e., object 4 is not a cutpoint, because it is pointed to by an actual parameter. Notice that here we have both stack sharing and heap sharing. The semantics treats both cases uniformly.

The two right stores represent impossible situations excluded by having a cutpoint. In memory state $s_G^{cp_{var}\,3.3}$ there is a cutpoint due to a local variable. In memory state $s_G^{cp_{fld}\,3.3}$ there is a cutpoint due to a field.

3.4 $\mathcal{LSL}^{\text{CPF}}$: A Localized-Heap Storeless Cutpoint-Free Semantics

In this section, we present $\mathcal{LSL}^{\text{CPF}}$, the Localized-heap Store-Less Cutpoint-Free semantics and investigate its properties. Similarly to the $\mathcal{GSB}$ semantics and the $\mathcal{LSB}$ semantics (see Sections 2.2 and 2.3, respectively), $\mathcal{LSL}^{\text{CPF}}$ is a natural semantics [Kah87]. However, $\mathcal{LSL}^{\text{CPF}}$ is a storeless semantics, i.e., memory cells are not identified by locations. Thus, we cannot talk about locations as in Sections 2.2 and 2.3. Instead, we use the term *objects*.[6]

To define the semantics, we use the function $\cdot\cdot$, defined in Figure 3.7. It is used as an infix operator. The application $\alpha.\delta$ concatenates the sequence of field identifiers δ to α. We say that an access path α is a *prefix* of an access path β, denoted by $\alpha \leq \beta$, when there is a field path $\delta \in \Delta$, such that $\beta = \alpha.\delta$. We say that α is a *proper prefix* of β, denoted by $\alpha < \beta$, when $\delta \neq \epsilon$. The function $\cdot\cdot$ is lifted to handle sets of access paths and sets of sequences of field identifiers.

[5] We note that the variant program can be effectively analyzed by the more general framework described in Chapter 4.

[6] We remind the reader that we keep making the simplifying assumptions listed in Section 2.2.1.

r	$\in$	$Root_p = V_p$	Roots of access paths
α, β	$\in$	$AccPath_p = Root_p \times \Delta$	Access paths
o	$\in$	$Obj^p_{L_{CPF}} = 2^{AccPath_p}$	Objects
A, A_p	$\in$	$Heap^p_{L_{CPF}} = 2^{Obj^p_{L_{CPF}}}$	Heaps
$\sigma_{L_{cpf}}, \sigma^p_L, \langle A_p \rangle$	$\in$	$\Sigma^p_{L_{CPF}} = Heap^p_{L_{CPF}}$	Memory states

Figure 3.4: Semantic domains of memory states for procedure p in $\mathcal{LSL}^{\text{CPF}}$. We use the syntactic domains V_p and $AccPath_p$ as semantic domains, too (and use italics font to denote a semantics value.)

In addition, we make use of the *flat* functional, well-known from functional programming. *flat* M returns the set of all elements of M, if M is a set of sets. Formally, *flat* $M \stackrel{\text{def}}{=} \{x \mid \exists A \in M : x \in A\}$.

3.4.1 Memory States

In this section, we define the representation of memory states in $\mathcal{LSL}^{\text{CPF}}$. Traditionally, a storeless semantics represents the heap by an equivalence relation over a set of access paths, where equivalence classes (implicitly) represent allocated objects. For readability, we use the equivalence classes directly.

Figure 3.4 defines the semantic domains used in $\mathcal{LSL}^{\text{CPF}}$ and the meta-variables ranging over them. A *memory state* $\sigma^p_L = \langle A_p \rangle$ for a procedure p is a single-element tuple containing a heap, denoted by A_p.[7] A heap $A_p \in Heap^p_{L_{CPF}}$ is a finite (but unbounded) set of objects. An object, denoted by o, is described by a (possibly infinite) set of access paths *rooted* at local variables of p. (Specifically, $\emptyset \notin A_p$).

A memory state $\sigma^p_L = \langle A_p \rangle$ at a given point in an execution is composed of a representation of the heap (A_p) at that point in the execution. To exclude states that cannot arise in any program, we now define the notion of *admissible storeless memory states* which must satisfy certain conditions.[8]

Definition 3.4.1 (Admissible storeless memory states) *A storeless memory state $\langle A_p \rangle$ for a procedure p at a given point in an execution is* ***admissible*** *iff*

(i) An access path points-to (at most) one object, i.e., $\forall o, o' \in A_p$ if $o \neq o'$, then $o \cap o' = \emptyset$;
(ii) A_p is right-regular, i.e., $\forall o_1, o_2 \in A_p$ if $\alpha, \beta \in o_1$ and $\alpha.\delta \in o_2$ then $\beta.\delta \in o_2$; and
(iii) A_p is prefix-closed, i.e., if $\alpha.f \in$ flat A_p, then $\alpha \in$ flat A_p.

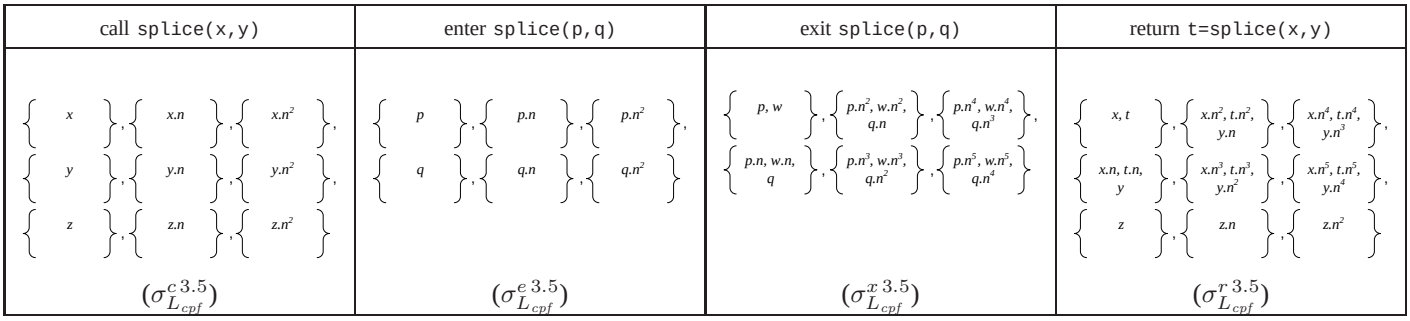
Because $\mathcal{LSL}^{\text{CPF}}$ preserves admissibility of states (see Section 3.5), in the sequel, whenever we refer to an $\mathcal{LSL}^{\text{CPF}}$ state, we mean an *admissible* $\mathcal{LSL}^{\text{CPF}}$ state.

Extraction of aliasing information. It is possible to extract aliasing relationships from the sets of access paths that describe the objects in a heap, and in particular to observe the heap structure as follows: a current variable `x` *points-to* an object o iff the access path $\langle x, \epsilon \rangle$ is in o. The field `f` of an object o_1 *points-to* object o_2 iff for every access path $\langle r, \delta \rangle$ in o_1, the access path $\langle r, \delta f \rangle$ is in o_2. An access path α *points-to* (resp. *passes through*) an object o, if $\alpha \in o$ (resp. $\exists \beta < \alpha$ such that $\beta \in o$). An object o is *reachable* from a variable x, if there exists a field path $\delta \in \Delta$ such that $\langle x, \delta \rangle \in o$.

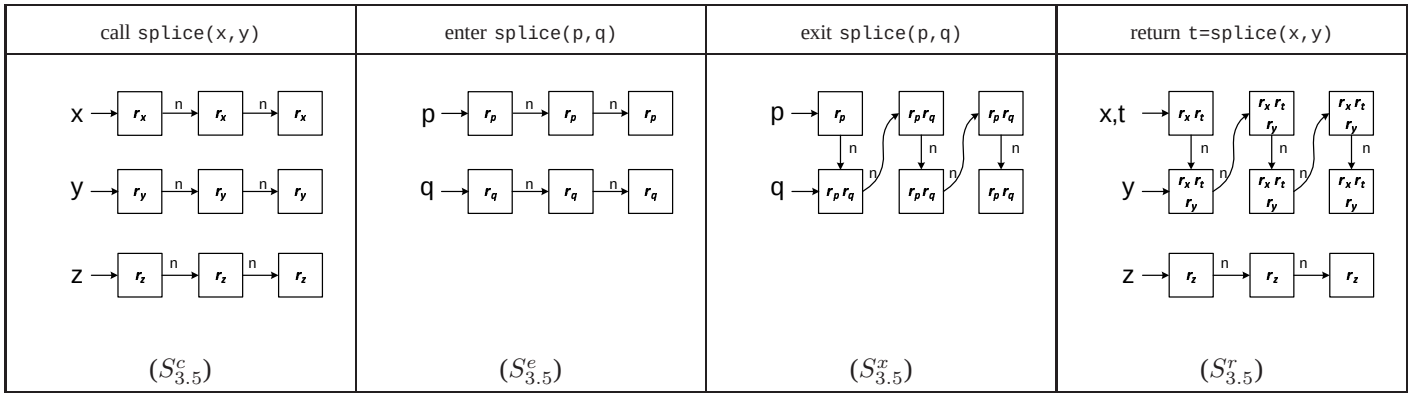
> **Example 3.4.2** Memory state $\sigma^{c\,3.5}_{L_{cpf}}$, shown at Figure 3.5(a), depicts the memory state of our running example when `splice` is invoked for the first time according to the $\mathcal{LSL}^{\text{CPF}}$ semantics. It shows nine sets of access paths. Each set represents one allocated list-element. At $\sigma^{c\,3.5}_{L_{cpf}}$, each object is pointed to by exactly one access path. For example, the first element, second element, and third element at `x`'s list, are represented by the sets of access paths $\{x\}$, $\{x.n\}$, and $\{x.n.n\}$, respectively. In particular, this invocation of `splice` is cutpoint-free.

[7] A memory state is considered to be a single-element tuple for presentation reasons only. Specifically, it helps elucidates the distinctions between the memory states of the $\mathcal{LSL}^{\text{CPF}}$ semantics and of the $\mathcal{LSL}$ semantics, defined in Chapter 4.

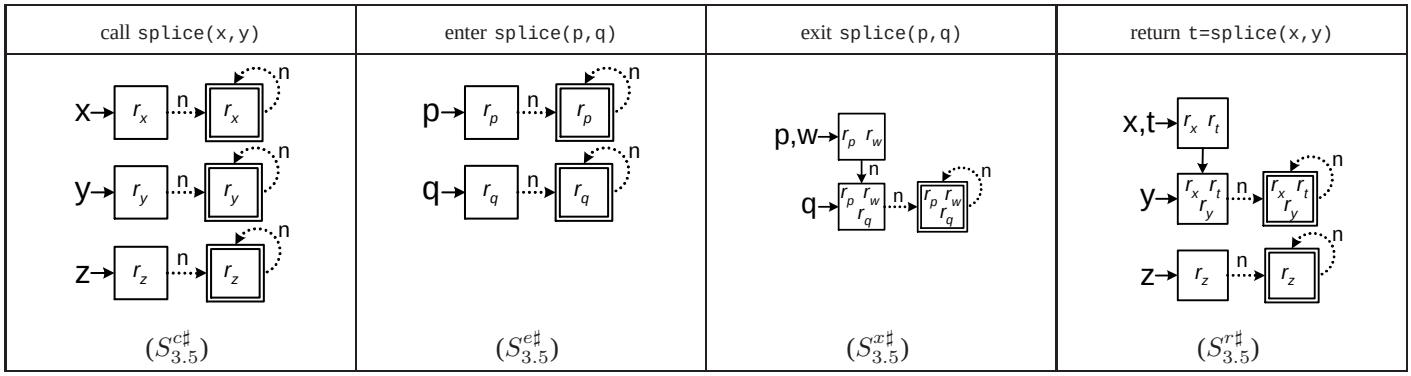
[8] These conditions are standard in storeless semantics. See, e.g., [Jon81, Deu92b, BIL03].



(a) Concrete states for the invocation `t = splice(x, y)` in the running example according to the $\mathcal{SL}^{\text{CPF}}$ semantics. We use $x.n^k$ as a shorthand for an access path rooted at `x` and traversing k `n`-fields. For example, we use $x.n^3$ as a shorthand for $x.n.n.n$.

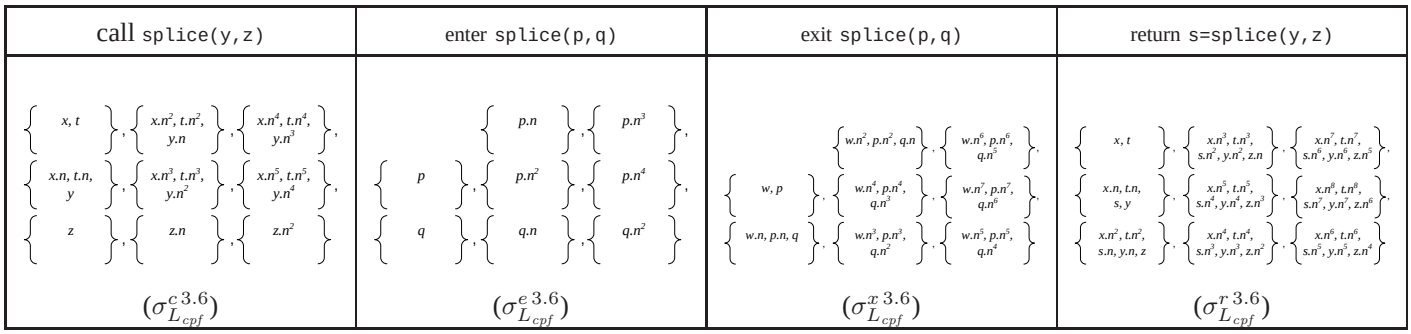


(b) The 2-valued logical structure that results by applying *to2VLS*^{cpf} to the memory states shown at Figure 3.5(a).

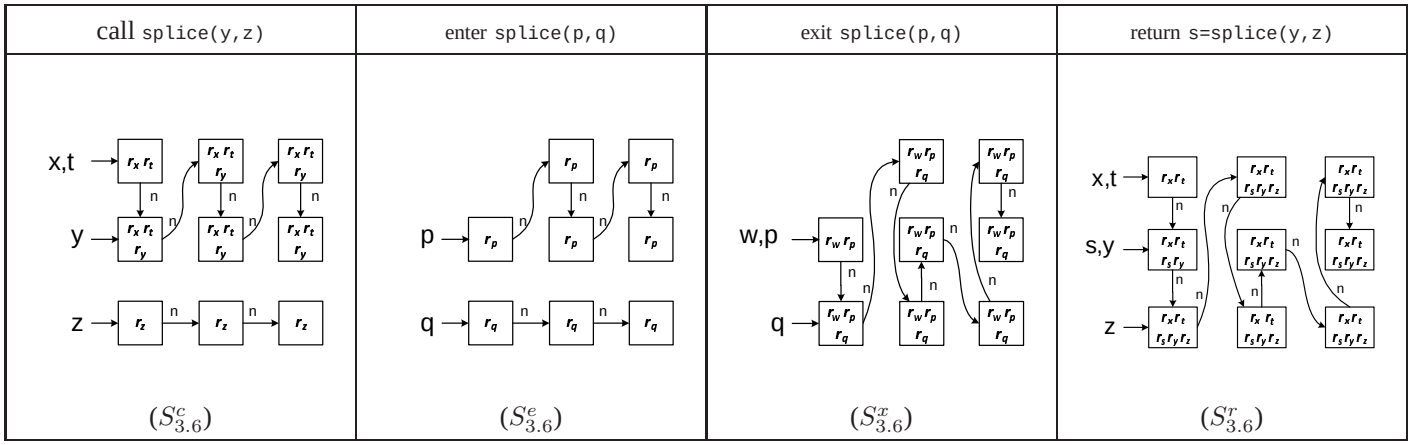


(c) The 3-valued logical structure that results by applying *canonical abstraction* to the 2-valued logical structures shown at Figure 3.5(b).

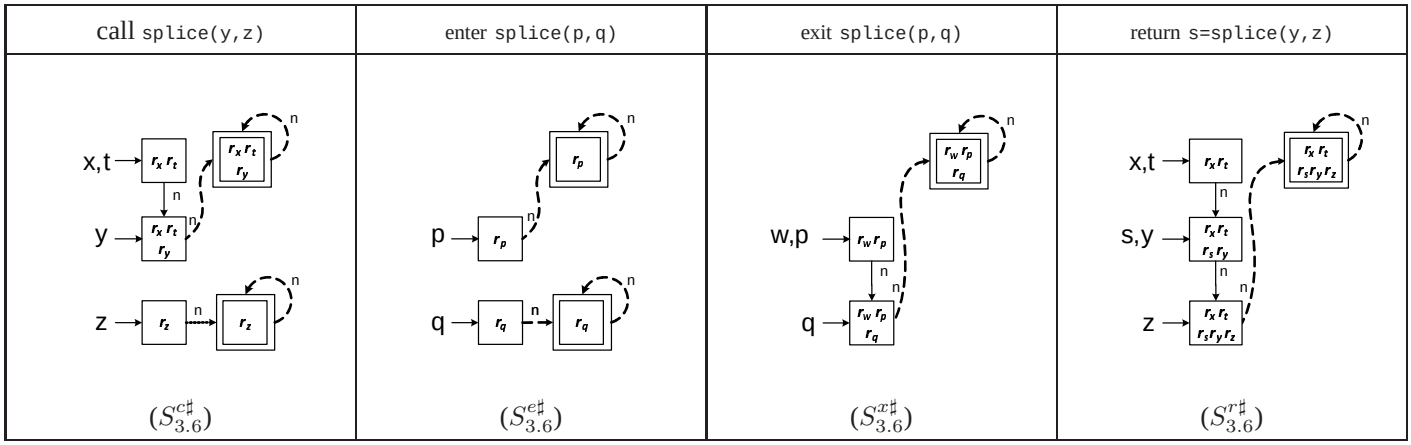
Figure 3.5: Memory states that may arise at the call-site, entry-site, exit-site, and return-site in the invocation `t = splice(x, y)` in the running example according to (a) the $\mathcal{SL}^{\text{CPF}}$ semantics, (b) the $\mathcal{LCPF}$ semantics, and (c) the $\mathcal{LCPF}^{\sharp}$ semantics.



(a) Concrete states for the invocation `s = splice(y, z)` in the running example according to the $\mathcal{LSL}^{\text{CPF}}$ semantics. We use $x.n^k$ as a shorthand for an access path rooted at `x` and traversing k `n`-fields. For example, we use $x.n^3$ as a shorthand for $x.n.n.n$.



(b) The 2-valued logical structure that results by applying *to2VLS*^{cpf} to the memory states shown at Figure 3.6(a).



(c) The 3-valued logical structure that results by applying *canonical abstraction* to the 2-valued logical structures shown at Figure 3.6(b).

Figure 3.6: Memory states that may arise at the call-site, entry-site, exit-site, and return-site in the invocation `s = splice(y, z)` in the running example according to (a) the $\mathcal{LSL}^{\text{CPF}}$ semantics, (b) the $\mathcal{LCPF}$ semantics, and (c) the $\mathcal{LCPF}^{\sharp}$ semantics.

$.: AccPath \times \Delta \to AccPath$ s.t.
$\langle r, \delta \rangle.\delta' \stackrel{\text{def}}{=} \langle r, \delta\delta' \rangle$

$.: 2^{AccPath} \times \Delta \to 2^{AccPath}$ s.t.
$a.\delta \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a\}$

$.: 2^{AccPath} \times 2^{\Delta} \to 2^{AccPath}$ s.t.
$a.D \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a, \delta \in D\}$

$[]: AccPath \times Heap_{L_{CPF}} \to Obj_{L_{CPF}}$ s.t.
$[\alpha]_A \stackrel{\text{def}}{=} \{\beta \in a \mid a \in A, \alpha \in a\}$

$rem: Heap_{L_{CPF}} \times 2^{AccPath} \to Heap_{L_{CPF}}$ s.t.
$rem(A, a) \stackrel{\text{def}}{=} (map(\lambda o.o \setminus a.\{\delta \in \Delta\})\ A) \setminus \{\emptyset\}$

$add: Heap_{L_{CPF}} \times 2^{AccPath} \times AccPath \to Heap_{L_{CPF}}$ s.t.
$add(A, a, \alpha) \stackrel{\text{def}}{=} map(\lambda o.\, o \cup a.\{\delta \in \Delta \mid \alpha.\delta \in o\})\ A$

Figure 3.7: Helper functions.

Memory state $\sigma^{e\,3.5}_{L_{cpf}}$, shown at Figure 3.5(a), depicts the memory state at the entry to the first invocation of `splice`. It differs from $\sigma^{c\,3.5}_{L_{cpf}}$ in two ways: (i) there are only six objects in the heap and (ii) objects are represented in terms of the access paths that start either with p or q, the formal parameters of `splice`.

3.4.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{LSL_{CPF}}{\rightsquigarrow} \subseteq (\Sigma_{L_{CPF}} \times \mathit{stms}) \times \Sigma_{L_{CPF}}$. We give axioms for assignments and an inference rule for procedure calls in Figure 3.8 and Figure 3.9, respectively. All other statements are handled in the standard way. (See, e.g., [Kah87, NNH99]. Also, see Section C.1.) To simplify notation, we assume A with a certain index (resp. prime) to be the heap component of a state $\sigma_{L_{cpf}}$ with the same index (resp. prime).

3.4.2.1 Helper Functions

To define the inference rules, we use the following functions: $[\cdot]_\cdot$, *rem*$(\cdot,\cdot)$ and *add*$(\cdot,\cdot)$, which are defined in Figure 3.7. We use a as a metavariable ranging over sets of access paths, which are not necessarily objects, whereas o always stands for objects.

The function $[\alpha]_A$ returns the object that α points-to in heap A. When α does not point-to any object, $[\alpha]_A$ returns the empty set (which by definition never describes an object pointed-to by a current, or even a pending, access path).

The function *rem* takes as its arguments a heap A and a set of access paths a. It removes from the description of every object in heap A all the access paths that have a prefix in a. Whenever *rem* removes all the access paths from the description of an object, that object is removed from the description of the heap. The function $add(A, a, \alpha)$ yields a modified version of heap A, where to every object $o \in A$ reachable from α by following some field path $\delta \in \Delta$, the access paths $a.\delta$ are added.

In addition, we make use of $map()$, another well known functional from functional programming. The functional $map(f)\ M$ applies f to every element of M and returns the resulting set. Formally, $map(f)\ M \stackrel{\text{def}}{=} \{f(x) \mid x \in M\}$.

$$\langle \texttt{x} = \texttt{alloc}\ t, \langle A \rangle \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \langle A \cup \{\{x\}\} \rangle$$

$$\langle \texttt{x} = \texttt{y}, \langle A \rangle \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \langle add(A, \{x\}, y) \rangle$$

$$\langle \texttt{x} = \texttt{null}, \langle A \rangle \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \langle rem(A, \{x\}) \rangle$$

$$\langle \texttt{x} = \texttt{y.f}, \langle A \rangle \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \langle add(A, \{x\}, y.f) \rangle \qquad y \in \mathit{flat}\ A$$

$$\langle \texttt{x.f} = \texttt{null}, \langle A \rangle \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \langle rem(A, [x]_A.f) \rangle \qquad x \in \mathit{flat}\ A$$

$$\langle \texttt{x.f} = \texttt{y}, \langle A \rangle \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \langle \overline{add(A, [x]_A.f, y)} \rangle \qquad x \in \mathit{flat}\ A$$

Figure 3.8: Axioms for atomic statements in the $\mathcal{LSL}^{\text{CPF}}$. The side-condition $x \in \mathit{flat}\ A$ (resp. $y \in \mathit{flat}\ A$) means that x's (resp. y's) value is not *null*.

3.4.2.2 Atomic Statements

The *axioms* for atomic statements are given in Figure 3.8 and explained below.[6]

The semantics of memory allocation `x = alloc t` adds a new object that is described by $\{x\}$ to the heap. Note that this definition (implicitly) initializes the fields of the new object to *null*. Also note that in a storeless semantics the semantics of object allocation is deterministic (cf. the semantics for object allocation in $\mathcal{GSB}$ and $\mathcal{LSB}$, our store-based semantics, defined in Figure 2.4.)

The semantics for the assignment `x = y` copies the value of the variable y into x by adding an access path $\langle x, \delta \rangle$ to any object o that can be reached from y by following a field path δ, i.e., $\langle y, \delta \rangle$ points-to o. This is accomplished by applying *add* to the given heap, the singleton set $\{x\}$, and the access path y.

Assigning `null` to a variable x does not modify the link structure of the heap. We only need to eliminate all the access paths that start with x, using the *rem* function.

The rule for field dereference `x = y.f` is similar. It adds the access path $\langle x, \delta \rangle$ to any object that can be reached from y by following field f, and then continuing with field path δ. Note, however, that the rule can be applied only if y points-to an object, i.e., the semantics checks that a null-dereference is not performed.

A destructive update `x.f = null` (potentially) modifies the link structure of the heap. Thus, every access path that has a prefix aliased with $\langle x, f \rangle$ is removed from the description of every object in the heap. Note, that $[x]_A$ returns all the access paths that are aliased with x. Concatenating $[x]_A$ with f returns the set of prefixes of affected access paths. Again, the rule can be applied only if x points-to an object.

An assignment `x.f = y` also has a (potential) effect on all the access paths that are aliased with x. After this assignment, any object o that can be reached by following the field path δ from y, i.e., $\langle y, \delta \rangle \in o$, is also reachable by traversing some access path aliased with x, followed by an f-field, and continuing with δ. As this is a place where cycles can be created, *add* does not necessarily return a right-regular heap (see Definition 3.4.1). Therefore, we apply the operator $\bar{\cdot}$. $\bar{A}$ is defined to be the set of equivalence classes obtained from the least right-regular, prefix-closed, equivalence relation that is a superset of the equivalence relation induced by A.[9] Note that this definition may only add access paths to the description of existing objects.

> **Example 3.4.3** Consider the memory state $\sigma^0_{L_{cpf}} = \langle \{\} \rangle$ in which the heap is empty. Executing the statement `y = alloc List` results in memory state $\sigma^1_{L_{cpf}} = \langle \{\{y\}\} \rangle$ containing a single allocated node object which is pointed to by variable y.
>
> Executing the statement `z = alloc List` on $\sigma^1_{L_{cpf}}$ results in memory state $\sigma^2_{L_{cpf}} = \langle \{\{y\}, \{z\}\} \rangle$ containing two allocated node objects, one is pointed to by y and the other is pointed to by z.
>
> Executing the statement `x = y` on $\sigma^2_{L_{cpf}}$ results in memory state $\sigma^3_{L_{cpf}} = \langle \{\{y, x\}, \{z\}\} \rangle$ in which the access path x is added to the representation of the node pointed to by y.
>
> Executing the statement `y.n = z` on $\sigma^3_{L_{cpf}}$ results in memory state $\sigma^4_{L_{cpf}} = \langle \{\{y, x\}, \{z, y.n, x.n\}\} \rangle$ in which the access paths $y.n$ *and* $x.n$ are added to the representa-

[9] The operator $\bar{\cdot}$ is similar to the ρ_{rstc} operator in [Deu92b].

tion of the node pointed to by z. (Note that $[y]_{\sigma^4_{L_{cpf}}} = \{y, x\}$, thus $[y]_{\sigma^4_{L_{cpf}}}.n = \{y.n, x.n\}$. Also note that no cycles are created in the heap. Thus, the resulting heap is right-regular).

Executing the statement `y = null` on $\sigma^4_{L_{cpf}}$ results in memory state $\sigma^5_{L_{cpf}} = \langle\{\{x\}, \{z, x.n\}\}\rangle$ in which any access path starting with y is removed from the representation of every object.

Executing the statement `x.n = null` on $\sigma^5_{L_{cpf}}$ removes the n-field between the object pointed to by x and the object pointed to by z, and results in memory state $\sigma^6_{L_{cpf}} = \langle\{\{x\}, \{z\}\}\rangle$ containing two allocated node objects, one is pointed to by x and the other is pointed to by z.

Example 3.4.4 Consider a memory state $\langle\{\{x\}\}\rangle$ containing a single allocated node object which is pointed to by variable x. Executing the statement `x.n = x` on this memory state results in a memory state in which the n-successor of the object pointed to by x points to itself, thus creating a cycle. Note that applying $add(\{\{x\}\}, [x]_{\{\{x\}\}}.n, x)$ results in $\{\{x, x.n\}\}$, and that using the operator $\bar{\cdot}$, we get the final description of the heap $\overline{\{\{x, x.n\}\}} = \{\{x, x.n, x.n.n, \ldots\}\}$.

3.4.2.3 Procedure Calls

The *inference rule* for procedure calls is defined in Figure 3.9. The rule defines the program state $\sigma^r_{L_{cpf}}$ that results from an invocation `y=p(x1,...,xk)` at memory state $\sigma^c_{L_{cpf}}$, assuming that the execution of the body of p at memory state $\sigma^e_{L_{cpf}}$ results in memory state $\sigma^x_{L_{cpf}}$. The heaps A^c and A^r are described by sets of access paths starting at the caller's variables, whereas the heaps A^e and A^x are described by sets of access paths that start at the callee's formal parameters and return variable. The rule provides the means to reconcile the different representations.

Our treatment of procedure call and return could be briefly described as follows:

(i) the call rule is applied, first checking that the invocation is cutpoint-free (by evaluating the side condition);
(ii) the memory state at the callee's entry site ($\sigma^e_{L_{cpf}}$) is constructed if the side condition holds; and
(iii) the caller's memory state at the call site ($\sigma^c_{L_{cpf}}$) and the callee's memory state at the exit site ($\sigma^x_{L_{cpf}}$) are used to construct the caller's memory state at the return site ($\sigma^r_{L_{cpf}}$).

The main idea behind the rule is to utilize the fact that a procedure cannot modify objects that are not in its local-heap (i.e., in the part of the heap that is *not* reachable from any actual parameter when the procedure is invoked). In particular, because $\mathcal{LSL}^{\text{CPF}}$ describes objects in terms of the access paths that point-to them, these "inaccessible" objects have the same description before and after the call. Thus, only the description of the objects in the function's local-heap (i.e., in the part of the heap that the procedure can access) is (possibly) updated. The update, is carried under the assumption that the invocation is cutpoint-free.[10] This restriction is checked at the call site by the side condition of the call rule.

Technically, the rule uses the functions $cpfCall_q^{y=p(x_1,\ldots,x_k)}$ and $cpfRet_q^{y=p(x_1,\ldots,x_k)}$, which are parameterized for each call statement in the program. $cpfCall_q^{y=p(x_1,\ldots,x_k)}$ computes the memory state $\sigma^e_{L_{cpf}}$ that results at the entry of p, the callee, when $y = p(x_1, \ldots, x_k)$ is invoked by q in memory state $\sigma^c_{L_{cpf}}$. The caller's memory state after the invocation is restored by the function $cpfRet_q^{y=p(x_1,\ldots,x_k)}$. This function computes the memory state of q, the caller, at the return-site ($\sigma^r_{L_{cpf}}$) according to q's memory state at the call-site ($\sigma^c_{L_{cpf}}$) and p's memory state at the exit-site ($\sigma^x_{L_{cpf}}$). In the rest of this section we describe the rule for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. The rule utilizes additional helper functions, defined in Figure 3.10, which we gradually explain. We now formally define and explain the way these functions implement the aforementioned three steps.

Verifying Cutpoint-Freedom

The semantics uses the side condition of the procedure call rule to ensure that the execution is cutpoint-free. The side condition asserts that no object is a cutpoint. Specifically, it computes the set *Cutpoints* of possible *cutpoints of the invocation*, and checks that it is empty.

The set *Cutpoints* is computed using the auxiliary function $CPObjs_q^{cpf}$, defined in Figure 3.10. The auxiliary function $CPObjs_q$ determines the cutpoints for this procedure invocation. Its parameters are the caller's (i.e.,

[10] The same mechanism is used to compute the description of objects that the callee allocates.

$$\frac{\langle \text{body of } p, \sigma^e_{L_{cpf}} \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \sigma^x_{L_{cpf}}}{\langle y = p(x_1, \ldots, x_k), \sigma^c_{L_{cpf}} \rangle \overset{LSL_{CPF}}{\rightsquigarrow} \sigma^r_{L_{cpf}}} \quad Cutpoints = \emptyset$$

where

$\sigma^e_{L_{cpf}} = \mathit{cpfCall}^{y=p(x_1,\ldots,x_k)}_q(\sigma^c_{L_{cpf}})$
$\sigma^r_{L_{cpf}} = \mathit{cpfRet}^{y=p(x_1,\ldots,x_k)}_q(\sigma^c_{L_{cpf}}, \sigma^x_{L_{cpf}})$
$Cutpoints = CPObjs^{cpf}_q(\langle A^c \rangle)\ (O^{args}_c, O^{passed}_c)$

$O^{args}_c = \mathit{PTo}(\{x_1, \ldots, x_k\})\ A^c$
$O^{passed}_c = \mathit{RObjs}(A^c)\ O^{args}_c$
$bind_{args} = \lambda o \in O^{args}_c.\{\langle h_i, \epsilon \rangle \mid 1 \leq i \leq k,\ x_i \in o\}$
$bind_{call} = bind_{args}$

$cpfCall^{y=p(x_1,\ldots,x_k)}_q : \Sigma^q_{L_{CPF}} \to \Sigma^p_{L_{CPF}}\ s.t.$
$cpfCall^{y=p(x_1,\ldots,x_k)}_q(\langle A^c \rangle) \overset{\text{def}}{=} \langle map(sub(bind_{call}))\ O^{passed}_c \rangle$

$cpfRet^{y=p(x_1,\ldots,x_k)}_q : \Sigma^q_{L_{CPF}} \times \Sigma^p_{L_{CPF}} \to \Sigma^q_{L_{CPF}}\ s.t.$
$cpfRet^{y=p(x_1,\ldots,x_k)}_q(\langle A^c \rangle, \langle A^x \rangle) \overset{\text{def}}{=} \langle (A^c \setminus O^{passed}_c) \cup map(sub(bind_{ret}))\ A^x \rangle$
where
$bind_{ret} = \lambda a \in range(bind_{call}) \cup \{\{\langle ret, \epsilon \rangle\}\}.$
$$\begin{cases} \{\langle y, \epsilon \rangle\} & a = \{\langle ret, \epsilon \rangle\} \\ Bypass(O^{passed}_c) \circ bind^{-1}_{call}(a) & \text{otherwise} \end{cases}$$

Figure 3.9: The inference rule for procedure calls in $\mathcal{LSL}^{\text{CPF}}$. The rule is given for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q. We assume that the formal parameters of p are $h_1, \ldots, h_k$. We recall that according to our conventions, $\sigma^c_{L_{cpf}} = \langle A^c \rangle$, $\sigma^e_{L_{cpf}} = \langle A^e \rangle$, $\sigma^x_{L_{cpf}} = \langle A^x \rangle$, and $\sigma^r_{L_{cpf}} = \langle A^r \rangle$.

$$
\begin{array}{l}
\textit{PTo}\colon 2^{\mathcal{V}} \to \mathit{Heap}_{L_{CPF}} \to 2^{Obj_{L_{CPF}}} \text{ s.t.} \\
\quad \textit{PTo}(V)\ A \stackrel{\text{def}}{=} \{[x]_A \neq \emptyset \mid x \in V\} \\
\\
\textit{RObjs}\colon \mathit{Heap}_{L_{CPF}} \to (2^{Obj_{L_{CPF}}} \to 2^{Obj_{L_{CPF}}}) \text{ s.t.} \\
\quad \textit{RObjs}(A)\ O \stackrel{\text{def}}{=} \{o \in A \mid o' \in O,\, \delta \in \Delta,\, o'.\delta \subseteq o\} \\
\\
\mathit{Bypass}\colon 2^{Obj_{L_{CPF}}} \to (Obj_{L_{CPF}} \to 2^{\textit{AccPath}}) \text{ s.t.} \\
\quad \mathit{Bypass}(O)\ o \stackrel{\text{def}}{=} \{\langle r, \delta\rangle \in o \mid \forall \delta' < \delta.\, \langle r, \delta'\rangle \notin \mathit{flat}\, O\} \\
\\
\mathit{sub}\colon (2^{AccPath} \to 2^{AccPath}) \to (Obj_{L_{CPF}} \to 2^{AccPath}) \text{ s.t.} \\
\quad \mathit{sub}(\mathit{bind})\ o \stackrel{\text{def}}{=} \mathit{flat} \left\{ \mathit{bind}(a).\delta \;\middle|\; \begin{array}{c} a \in \mathit{dom}(\mathit{bind}), \\ \delta \in \Delta,\, a.\delta \subseteq o \end{array} \right\} \\
\\
\mathit{CPObjs}_q^{cpf}\colon \Sigma^q_{L_{CPF}} \to (2^{Obj^q_{L_{CPF}}} \times 2^{Obj^q_{L_{CPF}}} \to 2^{Obj^q_{L_{CPF}}}) \text{ s.t.} \\
\quad \mathit{CPObjs}_q^{cpf}(\langle A^c\rangle)\ (O_c^{args}, O_c^{passed}) \stackrel{\text{def}}{=} \\
\quad\quad \mathit{Let} \\
\quad\quad\quad O_{deep} = O_c^{passed} \setminus O_c^{args} \\
\quad\quad\quad O_{vars} = \{[\langle x, \epsilon\rangle]_{A^c} \in O_{deep} \mid x \in V_q\} \\
\quad\quad\quad O_{fld} = \left\{ o \in O_{deep} \;\middle|\; \begin{array}{c} \exists o' \in A^c \setminus O_c^{passed}, \\ \exists f \in \mathcal{F},\, o'.f \subseteq o \end{array} \right\} \\
\quad\quad \mathit{in} \\
\quad\quad\quad O_{vars} \cup O_{fld}
\end{array}
$$

Figure 3.10: Helper functions for the procedure call rule. The function $CPObjs_q$ is parameterized for every function q in the program. Recall that V_q is the set of q's local variables.

procedure q's) heap at the call site (A^c), the objects pointed to by the actual parameters (O_c^{args}), and the relevant objects for the invocation (O_c^{passed}), i.e., the objects that are going to constitute the local-heap of the callee (i.e., procedure p). To find which objects are in the local-heap of the called procedure, i.e., the ones reachable from the actual parameters ($x_1, \ldots, x_k$), $\mathcal{LSL}^{\text{CPF}}$ computes the set of objects that are *pointed-to* by p's actual parameters (O_c^{args}). The auxiliary function *PTo* finds the objects pointed to by the caller's actual parameters (O_c^{args}). We refer to these objects as the *parameter objects of the invocation*. The auxiliary function *RObjs* finds the part of the caller's heap (A^c) that is reachable from the parameter objects of the invocation (O_c^{passed}), i.e., the relevant objects for the invocation.

Function $CPObjs_q$ determines the cutpoints for this procedure invocation, i.e., it computes the set of objects that "separate" O_c^{passed} from the rest of the caller's heap. Recall that we do not consider the parameter objects of the invocation[11] as cutpoints. Thus, $CPObjs_q$ considers only objects in $O_{deep} = O_c^{passed} \setminus O_c^{args}$ as possible cutpoints. Following the intuition of cutpoints as "separating objects", an object $o \in O_{deep}$ is qualified as a cutpoint if (and only if) one of the following holds:

- o is pointed-to by a local variable of the caller (O_{vars}), or
- o is pointed-to by an object in the part of the caller's heap that is not passed to the procedure (O_{fld}).

Note that it is not possible that o separates the heap of the *caller* from the heap of one of the pending calls, without being pointed to by a formal parameter of the caller (recall that formal parameters are not modified), because this would mean that o is a cutpoint of the invocation of the caller (O_{cpl}).

Example 3.4.5 The parameter objects in the invocation `t=splice(x, y);` in our running example are

$$O_c^{args} = \{\{x\}, \{y\}\}\,.$$

[11] We remind the reader that a parameter object of an invocation is an object which is pointed-to by an actual parameter.

The relevant objects for this invocation are

$$O_c^{passed} = \big\{\{x\},\{x.n\},\{x.n.n\},\{y\},\{y.n\},\{y.n.n\}\big\}\ .$$

The parameter objects in the invocation `t=splice(y, z);` in our running example are

$$O_c^{args} = \{\{y, x.n, t.n\},\{z\}\}\ .$$

The relevant objects for this invocation are

$$O_c^{passed} = \left\{ \begin{array}{ccc} & \{y.n, x.n^2, t.n^2\}, & \{y.n^3, x.n^4, t.n^4\}, \\ \{y, x.n, t.n\}, & \{y.n^2, x.n^3, t.n^3\}, & \{y.n^4, x.n^5, t.n^5\}, \\ \{z\}, & \{z.n\}, & \{z.n.n\} \end{array} \right\}\ .$$

Computing the Memory State at the Entry Site

The memory state at the entry site to p (denoted by $\sigma^e_{L_{cpf}}$) represents the local-heap passed to p. It contains objects in $\sigma^c_{L_{cpf}}$ which are relevant for the invocation (O_c^{passed}). Function $bind_{args}$, defined in Figure 3.9, and function sub, defined in Figure 3.10, provide the means to reconcile the different representations of the heap by the caller (procedure q) and by the callee (procedure p).

Function $bind_{args}$ maps objects pointed-to by actual parameters to the set of "trivial" access paths that are made up of the corresponding formal parameters. Function sub computes the callee's heap in terms of access paths that start at its formal parameters. It uses function $bind_{call}$, which is identical to $bind_{args}$,[12] and replaces every access path that starts with an actual parameter $\langle x_i, \delta\rangle$ in the representation of a (relevant) object o by an access path $\langle h_i, \delta\rangle$ that starts with the corresponding formal parameter. (All other access paths are removed).

Computing the Memory State at the Return Site

The memory state at the return-site (denoted by $\sigma^r_{L_{cpf}}$) is constructed as a combination of the memory state in which p was invoked (denoted by $\sigma^c_{L_{cpf}}$) and the memory state at p's exit-site (denoted by $\sigma^x_{L_{cpf}}$). Informally, $\sigma^c_{L_{cpf}}$ provides the information about the (unmodified) irrelevant objects and $\sigma^x_{L_{cpf}}$ contributes the information about the destructive updates and allocations made during the invocation.

The main challenge in computing the effect of a procedure is relating the objects at the call-site to the corresponding objects at the return site. The fact that the invocation is cutpoint-free guarantees that the only references into the local-heap (of the callee) are references to objects referenced by an actual parameter. This allows us to reflect the effect of p into the local-heap of q by, essentially: (i) replacing the relevant objects in $\sigma^c_{L_{cpf}}$ with $\sigma^x_{L_{cpf}}$, the local-heap at the exit from p; (ii) inverting the substitution done at the call-site, i.e., replacing the root of every access path that starts with a formal parameter with the corresponding actual parameter, and handling the binding of the return value in a similar way.

More technically, the description of the objects after the call should account for the mutations (destructive updates) of the heap performed by the callee. However, because the invoked procedure cannot modify objects that it cannot access, it can only modify fields of objects in O_c^{passed}. Thus, to compute the (possibly) updated description of objects in O_c^{passed} (as well as of objects that the callee allocates) it is sufficient to have a description of every object in O_c^{passed} (and of every object allocated by the callee) comprised of the access paths that start at objects that separate O_c^{passed} from the rest of the caller's heap: When the procedure returns, we just replace any access path $\langle r_p, \delta_p\rangle$ in the description of every object in the heap of the callee (A^x) that starts at a "separating object" o', by access paths of the caller $\langle r_q, \delta_q\delta_p\rangle$ such that $\langle r_q, \delta_q\rangle$ points-to o', but does not pass through O_c^{passed} (and thus cannot be modified).

To handle the return of procedure p, we use an additional binding, $bind_{ret}$. This mapping is the inverse of $bind_{call}$ (hence getting back to the caller's representation of the object) composed with the function $Bypass(O_c^{passed})$, which filters out access paths (of the caller) that *pass through* the part of the heap that p had

[12] We use function $bind_{call}$ for expository reasons only. Specifically, it helps elucidate the differences between the procedure call rule of the $\mathcal{LSL}^{\text{CPF}}$ semantics and that of the $\mathcal{LSL}$ semantics, defined in Chapter 4.

access to (O_c^{passed}). In addition, it also takes care of replacing access paths starting with special variable ret with the same access paths starting with result variable y.

The new heap is called A^r. It is derived by removing from the heap at the call-site the passed objects (O_c^{passed}), plugging in the heap that results from evaluating p's body (A^x), and substituting the description of all the objects by applying $sub(bind_{ret})$ to every object in A^x.

Example 3.4.6 Applying the procedure call rule for the invocation `t = splice(x, y);` in our running example results in the following sets and mappings:

$$\begin{array}{lcl} O_c^{args} & = & \{\{x\},\{y\}\} \\ O_c^{passed} & = & \{\{x\},\{x.n\},\{x.n.n\},\{y\},\{y.n\},\{y.n.n\}\} \\ Cutpoints & = & \emptyset \\ bind_{args} & = & [\{x\} \mapsto \{p\}, \{y\} \mapsto \{q\}] \\ bind_{ret} & = & \{\{x\} \mapsto \{p\}, \{y\} \mapsto \{q\}, \{ret\} \mapsto \{t\}\} \end{array}$$

Example 3.4.7 Applying the procedure-call rule for the invocation `s = splice(y, z);` in our running example results in the following sets and mappings:[13]

$$\begin{array}{lcl} O_c^{args} & = & \{\{y, x.n, t.n\}, \{z\}\} \\ O_c^{passed} & = & \left\{ \begin{array}{ccc} & \{y.n, x.n^2, t.n^2\}, & \{y.n^3, x.n^4, t.n^4\}, \\ \{y, x.n, t.n\}, & \{y.n^2, x.n^3, t.n^3\}, & \{y.n^4, x.n^5, t.n^5\}, \\ \{z\}, & \{z.n\}, & \{z.n.n\} \end{array} \right\} \\ Cutpoints & = & \emptyset \\ bind_{args} & = & [\{y, x.n, t.n\} \mapsto \{p\}, \{z\} \mapsto \{q\}] \\ bind_{ret} & = & \{\{y, x.n, t.n\} \mapsto \{p\}, \{z\} \mapsto \{q\}, \{ret\} \mapsto \{s\}\} \end{array}$$

Example 3.4.8 Applying the procedure-call rule for the invocation `s = splice(t, z);` in the variant of our running example results in the following sets and mappings:

$$\begin{array}{lcl} O_c^{args} & = & \{\{x, t\}, \{z\}\} \\ O_c^{passed} & = & \left\{ \begin{array}{ccc} \{x, t\} & \{y.n, x.n^2, t.n^2\}, & \{y.n^3, x.n^4, t.n^4\}, \\ \{y, x.n, t.n\}, & \{y.n^2, x.n^3, t.n^3\}, & \{y.n^4, x.n^5, t.n^5\}, \\ \{z\}, & \{z.n\}, & \{z.n.n\} \end{array} \right\} \\ Cutpoints & = & \{y, x.n, t.n\} \\ bind_{args} & = & [\{x, t\} \mapsto \{p\}, \{z\} \mapsto \{q\}] \end{array}$$

Note that $\langle y, \epsilon \rangle \in O_{deep} = O_{passed} \setminus O_{args}$, thus $\{y, x.n, t.n\} \in Cutpoints$. Specifically, $\mathcal{LSL}^{\text{CPF}}$ is able to detect that this invocation is not cutpoint-free.

3.5 Properties of the Semantics

In this section, we investigate the properties of the $\mathcal{LSL}^{\text{CPF}}$ semantics. In particular, we show that $\mathcal{LSL}^{\text{CPF}}$ can detect cutpoint-freedom of executions. We also show that for cutpoint-free executions, $\mathcal{LSL}^{\text{CPF}}$ is *observationally sound* (see Section 3.5.1) with respect to the standard semantics. Thus, abstractions of $\mathcal{LSL}^{\text{CPF}}$ can be used to conservatively verify properties of programs with respect to the standard semantics.

3.5.1 Observational Soundness

The only means by which a program can observe a state is by access paths. To state the theorems, we need some preliminary definitions about access-path equality and observational equivalence. We use the same simplifying notational conventions as in Section 2.4.1. Specifically, note that in both semantics an access path is equal to `null` when it has a prefix which is equal to `null`.

[13] The notation $x.n^k$ is explained in the caption of Figure 3.5.

Definition 3.5.1 (Access path equality) *Access paths α and β are **equal** in a given state $\sigma_{L_{cpf}}$, denoted by $[\![\alpha = \beta]\!]^{cpf}_L(\sigma_{L_{cpf}})$, if $\forall a \in A.\, \alpha \in a \iff \beta \in a$. An access path α is **equal to null** in state $\sigma_{L_{cpf}}$, denoted by $[\![\alpha = \texttt{null}]\!]^{cpf}_L(\sigma_{L_{cpf}})$, if $\alpha \notin$ flat A.*

Definition 3.5.2 (Observational equivalence) *Let p be a procedure. The states $\sigma_{L_{cpf}} \in \Sigma^p_{L_{CPF}}$ and $s_G \in \mathcal{S}^p_G$ are **observationally equivalent**, denoted by $\sigma_{L_{cpf}} \cong s_G$, if for all $\alpha, \beta, \gamma \in AccPath_p$,*

(i) $[\![\alpha = \beta]\!]^{cpf}_L(\sigma_{L_{cpf}}) \Leftrightarrow [\![\alpha = \beta]\!]_G(s_G)$, *and*

(ii) $[\![\gamma = \texttt{null}]\!]^{cpf}_L(\sigma_{L_{cpf}}) \Leftrightarrow [\![\gamma = \texttt{null}]\!]_G(s_G)$.

We also define observational equivalence between states in $\mathcal{LSL}^{\text{CPF}}$ in the same way.

Theorem 3.5.3 is the main theorem in this chapter. It states that $\mathcal{LSL}^{\text{CPF}}$ (i) detects if an execution is cutpoint-free, and (ii) for cutpoint-free executions, it is equivalent to $\mathcal{GSB}$, in the sense that both behave equivalently w.r.t. termination, and that execution of statements preserves observational equivalence.

Error state. Before defining the theorem, we slightly modify the $\mathcal{LSL}^{\text{CPF}}$ semantics in the following way. The semantics of procedure calls, as defined in Section 3.4.2.3, dictates that the procedure call rule cannot be used when the invocation violates the cutpoint-restriction. We change this behavior by adding a specially designated error state, $\mathbb{E}^{cp}$, and assuming that the semantics goes into this state when it detects a violation of the cutpoint-freedom restriction. Once the semantics reaches the error state, it keeps on propagating it, i.e., $\langle st, \mathbb{E}^{cp}\rangle \overset{L^{cpf}}{\rightsquigarrow} \mathbb{E}^{cp}$ for every $st \in$ *stms*.

Theorem 3.5.3 (Observational Soundness) *Let P be a program. Let p be a procedure in P. Let $\sigma_{L_{cpf}} \in \Sigma^{cpfp}_L$ and $s_G \in \mathcal{S}^p_G$ be observationally equivalent states, i.e., $\sigma_{L_{cpf}} \cong s_G$. Let st be an arbitrary statement in p. The following holds:*

$$\langle st, s_G\rangle \overset{GSB}{\rightsquigarrow} s'_G \Rightarrow \langle st, \sigma_{L_{cpf}}\rangle \overset{LSL}{\rightsquigarrow} \sigma'_{L_{cpf}}.$$

Furthermore, either $\sigma'_{L_{cpf}} \cong s'_G$ or $\sigma'_{L_{cpf}} = \mathbb{E}^{cp}$.

Sketch of Proof: We prove that $\mathcal{LSL}^{\text{CPF}}$ is observationally sound with respect to $\mathcal{LSB}$. From this proof together with Theorem 2.4.10, which states that $\mathcal{LSB}$ is observationally equivalent with $\mathcal{GSB}$, the theorem follows immediately.

The proof that $\mathcal{LSL}^{\text{CPF}}$ is observationally sound with respect to $\mathcal{LSB}$ is done by induction on the shape of the derivation trees. Specifically, we show that for every derivation tree in $\mathcal{LSB}$ resp. $\mathcal{LSL}^{\text{CPF}}$ there is a corresponding derivation tree in $\mathcal{LSL}^{\text{CPF}}$ resp. $\mathcal{LSB}$ with the same shape and, furthermore, corresponding memory states are either isomorphic or that $\mathcal{LSL}^{\text{CPF}}$ has reached the error state.

The key point in proof is identifying that in cutpoint-free invocations, the references going into the callee's local-heap (i.e., local variables of the caller or reference fields of objects outside the local-heap) can only point to parameter objects (i.e., any such reference is an alias of one of the actual parameters). Thus, when the procedure invocation returns, and the objects in the callee's local-heap need to be described in terms of access paths starting at local variables of the caller, the new description of the objects can be obtained from their description according to the callee's formal parameters and the return value.[14]

Theorem 3.5.3 ensures that abstract interpretation algorithms of $\mathcal{LSL}^{\text{CPF}}$ can be used to

(i) verify cutpoint-freedom, i.e., they are applicable to arbitrary programs and do not require an a-priori classification of a program as being cutpoint-free, and

(ii) compute conservative results with *respect to the standard heap semantics*. For example, static analysis algorithms which are based on the $\mathcal{LSL}^{\text{CPF}}$ semantics can be used to (a) verify data-structure invariants that are expressed by access-path equalities at a program point; (b) assert the absence of *null*-valued pointer dereferences; and (c) detect memory leaks.

[14] An alternative proof for Theorem 3.5.3 can be done by using Theorem 4.5.3, stated in Section 4.5: Theorem 4.5.3 states that $\mathcal{LSL}$, the non standard semantics defined in Chapter 4, is observationally equivalent to $\mathcal{GSB}$. Based on Theorem 4.5.3, the proof of Theorem 3.5.3 is immediate, because, for cutpoint-free executions, $\mathcal{LSL}^{\text{CPF}}$ produces the same derivation trees as $\mathcal{LSL}$.

Predicate	Intended Meaning
$inUc(v)$	v originates from the caller's memory state at the call site
$inUx(v)$	v originates from the callee's memory state at the exit site

Figure 3.11: Predicates used to implement the procedure call inference rule.

Theorem 3.5.3 also ensures that $\mathcal{LSL}^{\text{CPF}}$ is *heap-modular* in the following sense: (i) a procedure has no effect on the observable properties of the unreachable part of the heap, and (ii) a procedure cannot observe its context, i.e., that the execution of the function body is not affected by the irrelevant state.

The heap-modularity of $\mathcal{LSL}^{\text{CPF}}$ is quite expected because our semantics does not allow for `cast` statements. Thus, a procedure can access (i.e., either observe or mutate) only these parts of the heap that it can reach by traversing the link structure of the heap.

3.5.2 Admissibility

The following lemma ensures that the $\mathcal{LSL}^{\text{CPF}}$ semantics preserves admissible states (see Definition 3.4.1).

Lemma 3.5.4 (Admissibility) *Let st be a statement and $\sigma_{L_{cpf}} \in \Sigma_{L_{CPF}}$ an admissible state. If $\langle \text{st}, \sigma_L \rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$ then σ'_L is also an admissible state.*

Sketch of Proof: Immediate by induction on the shape of the derivation tree.

3.6 A Shape Abstraction of $\mathcal{LSL}^{\text{CPF}}$

In this section, we define a shape abstraction of $\mathcal{LSL}^{\text{CPF}}$ using *canonical abstraction* [SRW02]. The new abstraction forms the basis for a new interprocedural shape-analysis algorithm for cutpoint-free programs, described in Sections 3.7 and 3.8.

Technically, 3-valued logical structures are used to represent unbounded memory states. The tracked properties are encoded as predicates.

We define a Galois connection between the powerset domain of $\mathcal{LSL}^{\text{CPF}}$ memory states and $3Struct$ using a *representation function* (see Section 2.5.1) $\beta_L^{CPF}\colon \Sigma_L^{cpf} \rightarrow 3Struct$ which maps a program state to its "most-precise representation" in $3Struct$. Function β_L^{CPF} is a composition of two functions: (i) $to2VLS^{cpf}\colon \Sigma_L^{cpf} \rightarrow 2Struct$, which maps a cutpoint-free local-heap $\sigma_{L_{cpf}} \in \Sigma_L^{cpf}$ to an unbounded *2-valued* logical structure S, and (ii) *canonical abstraction*$\colon 2Struct \rightarrow 3Struct$ which conservatively bounds S. (cf. function β_L^{LSB}, defined in Section 2.5). The Galois connection

$$(2^{\Sigma_L^{cpf}}, \alpha\colon 2^{\Sigma_L^{cpf}} \rightarrow 2^{3Struct}, \gamma\colon 2^{3Struct} \rightarrow 2^{\Sigma_L^{cpf}}, 2^{3Struct})$$

is defined as:

$$\alpha(CC) = \{\beta_L^{CPF}(\sigma_{L_{cpf}}) \mid \sigma_L \in CC\} \text{ and } \gamma(AA) = \{\sigma_{L_{cpf}} \in \Sigma_L^{cpf} \mid S^\sharp \in AA,\ \beta_L^{CPF}(\sigma_{L_{cpf}}) \sqsubseteq S^\sharp\},$$

where $\beta_L^{CPF}(\sigma_{L_{cpf}}) \sqsubseteq S^\sharp$ means that $S^\sharp \in 3Struct$ conservatively represents $\beta_L^{CPF}(\sigma_{L_{cpf}}) \in 2Struct$. (See Definition 2.5.4).

3.6.1 Representing Memory States of $\mathcal{LSL}^{\text{CPF}}$ by 2-Valued Logical Structures

Function $to2VLS^{cpf}$, defined in Figure 3.12, maps a memory state $\sigma_{L_{cpf}} = \langle A \rangle \in \Sigma_{L_{CPF}}$ to a *2-valued* logical structure S. Every object $o \in A$ is represented by a unique node in U^S. Tracked properties of the memory state are recorded by the predicates given in Figure 2.11, whose intended meaning is explained in Section 2.5.1.2. In addition, we use the predicates $inUc$ and $inUx$, shown in Figure 3.11 to implement the call rule. The role of these predicates is explained in Section 3.7.2.2.

We track properties of different procedures using different sets of predicates. To enable that, we assume that all structures are using the same global set of predicates $\mathcal{P}$, and assume that every procedure p is associated with a

to2VLS^{cpf} : $\Sigma_{L_{CPF}} \rightarrow$ *2-Struct* s.t.
to2VLS$^{cpf}(\langle A \rangle) = S$ where $S = \langle U^S, \iota^S \rangle$ and
$U^S = A$

$$
\begin{array}{lcl}
\iota^S(x)(v) & = & v \in A \text{ and } x \in v \\
\iota^S(n)(v_1, v_2) & = & v_1 \in A, v_2 \in A \text{ and } v_1.n \subseteq v_2 \\
\iota^S(eq)(v_1, v_2) & = & v_1 = v_2 \\
 & & \\
\iota^S(r_x)(v_1) & = & \exists \alpha \in v_1 \text{ s.t. } \langle x, \epsilon \rangle \leq \alpha \\
\iota^S(ils)(v) & = & \exists \alpha.n \in v,\ \beta.n \in v \text{ s.t. } [\alpha]_A \neq [\beta]_A \\
\iota^S(c)(v) & = & \exists \alpha \in v,\ \beta \in v \text{ s.t. } \alpha < \beta
\end{array}
$$

Figure 3.12: The function *to2VLS*^{cpf} maps states in Σ_L to 2-valued logical structures.

set of predicates $\mathcal{P}_p \subseteq \mathcal{P}$. The interpretation function of a structure $S = \langle U^S, \iota^S \rangle$ representing the memory state of a procedure p is a total function $\iota^S : \mathcal{P}_p \rightarrow \{0, \frac{1}{2}, 1\}$, i.e., ι^S defines (only) the meaning of the subset $\mathcal{P}_p \subseteq \mathcal{P}$ of predicates. (Note that ι^S is a *partial* function from $\mathcal{P}$).[15]

In the rest of this chapter, we assume to be working with a fixed arbitrary program P. The set of all reference fields defined in P is denoted by $FieldId^\star$. For a procedure p, V_p denotes the set of its local reference variables, including its formal parameters. The set of all the local (reference) variables in P is denoted by $Local^\star$. For simplicity, we assume formal parameters are not assigned and that p always returns a value using a designated variable $ret_p \in V_p$. For example, $ret_{\texttt{splice}} = \texttt{w}$.

Example 3.6.1 Figure 3.5(b) shows the 2-valued logical structures pertaining to the memory states at the call-site, entry-site, exit-site, and return-site during the invocation `t = splice(x, y)` in the running example, shown in Figure 3.5(a). Specifically, $S^c_{3.5} =$ *to2VLS*$^{cpf}(\sigma^{c\,3.5}_{L_{cpf}})$, $S^e_{3.5} =$ *to2VLS*$^{cpf}(\sigma^{e\,3.5}_{L_{cpf}})$, $S^x_{3.5} =$ *to2VLS*$^{cpf}(\sigma^{x\,3.5}_{L_{cpf}})$, and $S^r_{3.5} =$ *to2VLS*$^{cpf}(\sigma^{r\,3.5}_{L_{cpf}})$. (*2-valued* logical structures are depicted as directed graphs using the graphical notations introduced in Example 2.5.2).

Example 3.6.2 Figure 3.6(b) shows the 2-valued logical structures pertaining to the memory states at the call-site, entry-site, exit-site, and return-site during the invocation `s = splice(y, z)` in the running example, shown in Figure 3.6(a). Specifically, $S^c_{3.6} =$ *to2VLS*$^{cpf}(\sigma^{c\,3.6}_{L_{cpf}})$, $S^e_{3.6} =$ *to2VLS*$^{cpf}(\sigma^{e\,3.6}_{L_{cpf}})$, $S^x_{3.6} =$ *to2VLS*$^{cpf}(\sigma^{x\,3.6}_{L_{cpf}})$, and $S^r_{3.6} =$ *to2VLS*$^{cpf}(\sigma^{r\,3.6}_{L_{cpf}})$.

3.6.1.1 Admissible Memory States

Not all *2-valued* logical structures represent memory states that are compatible with the semantics of EAlgol (or JAVA or C, for that matter). For example, in EAlgol, each pointer variable points to at most one heap-allocated element. To exclude states that cannot arise in any program, we now define the notion of *admissible* 2*-valued logical structures*. This notion is similar to the notion of *admissible storeless memory states* (see Section 2.5.2.2) and to the notion of structures that are *compatible with hygiene conditions* in [SRW02].

Definition 3.6.3 (Admissible 2-Valued Logical Structures) *A* 2-valued *logical structure $S = \langle U, \iota \rangle$ representing a cutpoint-free local-heap for a procedure p at a given point in an execution is* ***admissible*** *iff*

(i) Only the local variables of the current call are represented, i.e., $\iota^S(x)$ is undefined for every $x \in Local^\star \setminus V_p$.
(ii) A variable points-to at most one node, i.e., for every $x \in V_p$, there exists at most one individual such that $\iota^S(x)(u) = 1$.
(iii) A field is a partial function, i.e., for every individual u_1 and every field $f \in FieldId^\star$, there exists at most one individual $u_2 \in U$ such that $\iota^S(f)(u_1, u_2) = 1$.
(iv) The predicate eq records equality, i.e., for every $u_1, u_2 \in U$, $\iota^S(eq)(u_1, u_2) = 1$ iff $u_1 = u_2$.

[15] Function *to2VLS*^{cpf}, as defined in Figure 3.12, uses, as an example, predicates which are suitable for representing memory states of procedures manipulating singly linked lists. In general, other predicates could have been used. In contrast, predicates $inUc$ and $inUx$ are never used to represent memory states. They are only used to define the return structure. See Section 3.7.2.2.

$$\langle st, S\rangle \overset{LCPF^\sharp}{\rightsquigarrow} \{\beta_L(\sigma'_{L_{cpf}}) \mid \sigma_{L_{cpf}} \in \gamma(S), \langle st, \sigma_{L_{cpf}}\rangle \overset{LSL_{CPF}}{\rightsquigarrow} \sigma'_{L_{cpf}}\}$$

Figure 3.13: A specification of the abstract inference rules for atomic statements.

3.6.2 Conservatively Representing Memory States of $\mathcal{LSL}^{\mathbf{CPF}}$ by 3-Valued Logical Structures using Canonical Abstraction

We obtain a *bounded* conservative representation of (unbounded) *2-valued* logical structures using canonical abstraction. (See Section 2.5.1.3.)

Example 3.6.4 Figure 3.5(c) depicts the 3-valued logical structure that results by applying *canonical abstraction* to the 2-valued logical structures representing the memory states at the call-site, entry-site, exit-site, and return-site in the first call to `splice` in the running example, shown in Figure 3.5(b). Specifically, $S^{c\sharp}_{3.5}$ is a canonical abstraction of $S^{c}_{3.5}$, $S^{e\sharp}_{3.5}$ is a canonical abstraction of $S^{e}_{3.5}$, $S^{x\sharp}_{3.5}$ is a canonical abstraction of $S^{x}_{3.5}$, and $S^{r\sharp}_{3.5}$ is a canonical abstraction of $S^{r}_{3.5}$. (*3-valued* logical structures are depicted using the graphical conventions introduced in Example 2.5.5.)

Example 3.6.5 Figure 3.6(c) depicts the 3-valued logical structure that results by applying *canonical abstraction* to the 2-valued logical structures representing the memory states at the call-site, entry-site, exit-site, and return-site in the second call to `splice` in the running example, shown in Figure 3.6(b). Specifically, $S^{c\sharp}_{3.6}$ is a canonical abstraction of $S^{c}_{3.6}$, $S^{e\sharp}_{3.6}$ is a canonical abstraction of $S^{e}_{3.6}$, $S^{x\sharp}_{3.6}$ is a canonical abstraction of $S^{x}_{3.6}$, and $S^{r\sharp}_{3.6}$ is a canonical abstraction of $S^{r}_{3.6}$. (*3-valued* logical structures are depicted using the graphical conventions introduced in Example 2.5.5.)

3.7 Abstract Transformers

In this section, we define the abstract transformers used by the analysis. In Section 3.7.1, we provide a declarative specification of the abstract transformers in a non-algorithmic fashion: We specify the abstract semantics using the *best abstract transformer* [CC79] (see Section 2.5.2.2). In Sections 3.7.2 and 3.7.3, we utilize the framework of [SRW02] to obtain conservative abstract transformers: We define the effect of intraprocedural statements as well as call and return statements using first order formulae with transitive closure and show how to compute the (abstract) effect of program statements.

3.7.1 A Declarative Specification of the Abstract Transformers

The meaning of statements is described by a transition relation $\overset{LCPF^\sharp}{\rightsquigarrow} \subseteq (3Struct \times st) \times 3Struct$. In this section, we provide a declarative specification of the meaning of statements, given by "abstract" inference rules in the same style as the natural semantics. The abstract inference rules operate on *3-valued* logical structures. (See Section 2.5.2.2). Figure 3.13 and Figure 3.14 show the specification of the abstract inference rules for atomic statements and procedure-calls, respectively. These rules are given in the declarative style of the best abstract transformer [CC79]: every abstract inference rule emulates a corresponding concrete inference rule using represented states (see Figure 2.16).

Example 3.7.1 Figure 3.6(c) shows an application of the procedure-call inference rule from Figure 3.14 to the second invocation of `splice` in our running example. The logical structures are: $S^{c\sharp}_{3.6}$, which arises at the call-site; $S^{e\sharp}_{3.6}$, which arises at the entry to `splice`; $S^{x\sharp}_{3.6}$, which arises at the exit-point of `splice`; and $S^{r\sharp}_{3.6}$, the structure *computed* at the return-site.

In $S^{x\sharp}_{3.6}$, the list pointed-to by p is spliced with the list pointed to by q. As a result, all the list elements are now reachable from p and q at the exit-site. Therefore, even though the list-element pointed-to by x is not explicitly represented in $S^{x\sharp}_{3.6}$, the inference rule allows us to conclude that at $S^{r\sharp}_{3.6}$, the logical structure at the return site, all the list elements become reachable from x. To conclude, definite values of many of the tracked properties of x can be established after the function call returns.

$$\frac{\langle \textit{body of } p, XS_p\rangle \overset{LCPF^\sharp}{\rightsquigarrow} XS'_p}{\langle y = p(x_1,\ldots,x_k), XS_q\rangle \overset{LCPF^\sharp}{\rightsquigarrow} XS'_q} \quad \textit{Cutpoints} = \emptyset$$

where

$$\{cpfCall_q^{y=p(x_1,\ldots,x_k)}(\sigma_L^c) \mid \sigma_{L_{cpf}}^c \in \gamma(XS_q)\} \subseteq \gamma(XS_p)$$

$$\left\{ cpfRet_q^{y=p(x_1,\ldots,x_k)}(\sigma_{L_{cpf}}^c, \sigma_{L_{cpf}}^x) \;\middle|\; \begin{array}{c} \sigma_{L_{cpf}}^c \in \gamma(XS_q), \\ \sigma_{L_{cpf}}^x \in \gamma(XS'_p), \\ compatible(\sigma_{L_{cpf}}^c, \sigma_{L_{cpf}}^x), \end{array} \right\} \subseteq \gamma(XS'_q)$$

$$compatible(\sigma_L^c, \sigma_L^x) \iff \left(\begin{array}{c} \forall h, h' \in F_p.[\![h = h']\!]_L^{cpf}(\sigma_{L_{cpf}}^e) \iff [\![h = h']\!]_L^{cpf}(\sigma_{L_{cpf}}^x) \wedge \\ \forall h \in F_p.[\![h = \texttt{null}]\!]_L^{cpf}(\sigma_{L_{cpf}}^e) \iff [\![h = \texttt{null}]\!]_L^{cpf}(\sigma_{L_{cpf}}^x)) \end{array} \right)$$

where $\sigma_{L_{cpf}}^e = cpfCall_q^{y=p(x_1,\ldots,x_k)}(\sigma_{L_{cpf}}^c)$.

$$\textit{Cutpoints} = \{o \in CPObjs_q^{cpf}(\langle A^c\rangle)\,(O_c^{args}, O_c^{passed}) \mid \langle A^c\rangle \in \gamma(XS_q)\}$$
$$O_c^{args} = \textit{PTo}(\{x_1,\ldots,x_k\})\,A^c$$
$$O_c^{passed} = \textit{RObjs}(A^c)\,O_c^{args}$$

Figure 3.14: A specification of the abstract inference rules for procedure calls. The functions $cpfCall_q^{y=p(x_1,\ldots,x_k)}$ and $cpfRet_q^{y=p(x_1,\ldots,x_k)}$ are defined in Figure 3.9. Note that we apply $cpfRet_q^{y=p(x_1,\ldots,x_k)}$ only for *compatible* pairs of memory states. Memory states σ_L^c and σ_L^x are compatible when they agree about the aliasing between and the nullnesss of formal parameters. (Recall that F_p denotes the set of the formal parameters of procedure p.) The rule requires that the invocation of p on any memory state represented at the call-site be cutpoint-free.

3.7.2 $\mathcal{LCPF}$: A Logic-based Concrete Localized-Heap Semantics for Cutpoint-Free Programs

The framework of [SRW02] allows to automatically derive the abstract transformers from a specification of the concrete semantics which uses logical formulae. Thus, we encode $\mathcal{LSL}^{\text{CPF}}$ in this form, Specifically, in this section we present $\mathcal{LCPF}$, a non-standard large-step operational semantics which implements $\mathcal{LSL}^{\text{CPF}}$ using logic formulae to specify the (concrete) meaning of statements.

$\mathcal{LCPF}$, like $\mathcal{LSL}^{\text{CPF}}$, is interesting because procedures operate on local-heaps, thus supporting the notion of heap-modularity while permitting the usage of a global-heap and destructive updates. $\mathcal{LCPF}$, like $\mathcal{LSL}^{\text{CPF}}$, checks that a program execution is indeed cutpoint-free and halts otherwise. As a result, $\mathcal{LCPF}$, like $\mathcal{LSL}^{\text{CPF}}$, is applicable to any arbitrary program, and does not require an a priori classification of a program as cutpoint-free. $\mathcal{LCPF}$ is defined to be observationally equivalent with $\mathcal{LSL}^{\text{CPF}}$. Thus, by construction, $\mathcal{LCPF}$ is observationally sound with respect to the standard global-heap semantics (see Theorem 3.5.3).

3.7.2.1 Concrete Memory States

We represent concrete memory states using 2-valued logical structures: each individual in the universe represents a heap-allocated object and every predicate corresponds to a tracked property of heap-allocated objects. (See Section 3.6.1).

3.7.2.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{LCPF}{\rightsquigarrow} \subseteq (2Struct \times st) \times 2Struct$ that specifies how a statement st transforms an incoming logical structure into an outgoing logical structure. For assignments, this is done primarily by defining the values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [SRW02]. The inference rules for assignments are

Shorthand	**Intended Meaning**	**Defining Formula**
$F(v_1, v_2)$	v_1 has a field that points to v_2	$\bigvee_{f \in FieldId^\star_P} f(v_1, v_2)$
$\varphi^*(v_1, v_2)$	the reflexive transitive closure of φ	$v_1 = v_2 \vee (TC\ w_1, w_2 : \varphi(w_1, w_2))(v_1, v_2)$
$R_{\{x_1,\ldots,x_k\}}(v)$	v is reachable from $\mathtt{x}_1$ or ... or $\mathtt{x}_k$	$\bigvee_{x \in \{x_1,\ldots,x_k\}} \exists v_1 : x(v_1) \wedge F^*(v_1, v)$
$isCP_{q,\{x_1,\ldots,x_k\}}(v)$	v is a cutpoint	$R_{\{x_1,\ldots,x_k\}}(v) \wedge$ $(\neg x_1(v) \wedge \ldots \wedge \neg x_k(v)) \wedge$ $\left(\bigvee_{y \in V_q} y(v) \vee \exists v_1 : \neg R_{\{x_1,\ldots,x_k\}}(v_1) \wedge F(v_1, v)\right)$

Figure 3.15: Formulae shorthands and their intended meaning. Recall that $FieldId^\star_P$ is the (necessarily bounded) set of the fields of types defined in program P. Thus, the formula $F(v_1, v_2)$, which holds when v_1 has a field pointing to v_2, is well defined. Formula $\varphi^*(v_1, v_2)$ uses the transitive closure operator TC, formally defined in Appendix A.2. Formulae $R_{\{x_1,\ldots,x_k\}}(v)$ holds when v is reachable from a local variable x in $\{x_1, \ldots, x_k\}$, i.e., v can be reached by following a (possibly empty) path of fields starting at the object pointed to by x. Formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$ is explained in Section 3.7.2.2.

$$\frac{\langle \textit{body of } p, S_e\rangle \overset{LCPF}{\rightsquigarrow} S_x}{\langle y = p(x_1,\ldots,x_k), S_c\rangle \overset{LCPF}{\rightsquigarrow} S_r} \qquad S_c \models \forall v\colon \neg isCP_{q,\{x_1,\ldots,x_k\}}(v)$$

where

$S_e = \langle U_e, \iota_e\rangle$ where
$U_e = \{u \in U^{S_c} \mid S_c \models R_{\{x_1,\ldots,x_k\}}(u)\}$
$\iota_e = updCall_q^{y=p(x_1,\ldots,x_k)}(S_c)$

$S_r = \langle U_r, \iota_r\rangle$ where
$Let\ \ U' = \{u.c \mid u \in U_c\} \cup \{u.x \mid u \in U_x\}$

$$\iota' = \lambda p \in \mathcal{P}.\begin{cases} \iota_c[inUc \mapsto \lambda v.1](p)(u_1,\ldots,u_m) & : \ u_1 = w_1.c, \ldots, u_m = w_m.c \\ \iota_x[inUx \mapsto \lambda v.1](p)(u_1,\ldots,u_m) & : \ u_1 = w_1.x, \ldots, u_m = w_m.x \\ 0 & : \ otherwise \end{cases}$$

$in\ \ U_r = \{u \in U' \mid \langle U', \iota'\rangle \not\models inUc(u) \wedge R_{\{x_1,\ldots,x_k\}}(u)\}$
$\iota_r = updRet_q^{y=p(x_1,\ldots,x_k)}(\langle U', \iota'\rangle)$

Figure 3.16: The inference rule for a procedure call $y = p(x_1, \ldots, x_k)$ by a procedure q. $S_c = \langle U_c, \iota_c\rangle$. The functions $updCall_q^{y=p(x_1,\ldots,x_k)}$ and $updRet_q^{y=p(x_1,\ldots,x_k)}$ are defined in Figure 3.17. The rule is explained in Section 3.7.2.2.

rather straightforward and can be found in Appendix C.2.1. For control statements, we use the standard rules of natural semantics, see, e.g., [Kah87, NNH99].

Our treatment of procedure call and return in $\mathcal{LCPF}$ follows their treatment in $\mathcal{LSL}^{\text{CPF}}$, and could be briefly described as follows: (i) the call rule is applied, first checking that the invocation is cutpoint-free (by evaluating the side condition), and (ii) proceeding to construct the memory state at the callee's entry site (S_e) if the side condition holds; (iii) the caller's memory state at the call site (S_c) and the callee's memory state at the exit site (S_x) are used to construct the caller's memory state at the return site (S_r). We now formally define and explain these steps.

Figure 3.16 specifies the procedure call rule for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q. The rule is instantiated for each call statement in the program, and gradually explained below.

Verifying Cutpoint-freedom

The semantics uses the side condition of the procedure call rule to ensure that the execution is cutpoint-free. The side condition asserts that no object is a cutpoint. This is achieved by verifying that the formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$, defined in Figure 3.15, does not hold for any object at S_c, the memory state that arises when $p(x_1, \ldots, x_k)$ is invoked by q.

a. Predicate update formulae for $updCall_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} x_i(v) & : \quad z = h_i \\ 0 & : \quad z \in V_p \setminus \{h_1, \ldots, h_k\} \end{cases}$$

b. Predicate update formulae for $updRet_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} ret_p(v) & : \quad z = y \\ inUc(v) \wedge z(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee \exists v_1 \colon z(v_1) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) & : \quad z \in V_q \setminus \{y\} \end{cases}$$

$$f'(v_1, v_2) = inUx(v_1) \wedge inUx(v_2) \wedge f(v_1, v_2) \vee inUc(v_1) \wedge inUc(v_2) \wedge f(v_1, v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{sep} \colon f(v_1, v_{sep}) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{sep}, v_2)$$

where $match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v_2) = \bigvee_{i=1}^{k} inUc(v_1) \wedge x_i(v_1) \wedge inUx(v_2) \wedge h_i(v_2)$

$$inUc'(v) = 0$$

$$inUx'(v) = 0$$

Figure 3.17: Predicate-update formulae for the core predicates used in the procedure call rule of $\mathcal{LCPF}$. We assume that the p's formal parameters are $h_1, \ldots, h_k$. There is a separate update formula for every local variable $\mathtt{z} \in Local^\star$ and for every field $\mathtt{f} \in FieldId^\star$. We remind the reader that according to our assumptions, procedure p always returns a value using the designated variable $ret_p \in V_p$.

The formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$, holding when v is a cutpoint object, is comprised of three conjuncts. The first conjunct, requires that v be reachable from an actual parameter. The second conjunct, requires that v not be pointed-to by an actual parameter. The third conjunct, requires that v be an entry point into p's local-heap, i.e., is pointed-to by a local variable of q (the caller procedure) or by a field of an object not passed to p.

Example 3.7.2 The structure $S^c_{3.5}$, shown at Figure 3.5, depicts the memory state at the point of the call `t = splice(x, y)`. In this state, the formula $isCP_{main,\{x,y\}}(v)$ does not hold for any object. Indeed, this invocation is cutpoint-free. Similarly, the invocation `s = splice(y, z)` on the memory state depicted by structure $S^c_{3.6}$, shown at Figure 3.6, is also cutpoint-free. Again, formula $isCP_{main,\{y,z\}}(v)$ does not hold for any object in this memory state. In particular, when v is bound to the object pointed to by `x` and `t`, the first conjunct of formula $isCP_{main,\{y,z\}}(v)$, i.e., $R_{\{y,z\}}(v)$, does not hold because this object is not reachable from either `y` or `z`. When v is bound to the object pointed to by `y`, formula $isCP_{main,\{y,z\}}(v)$ does not hold either, although formula $R_{\{y,z\}}(v)$ holds as well as the second disjunct in the third conjunct of $isCP_{main,\{y,z\}}(v)$, i.e., $\exists v_1 \colon \neg R_{\{y,z\}}(v_1) \wedge F(v_1, v)$. However this object is not a cutpoint because it is pointed to by the actual parameter `y`, as detected by the second conjunct of $isCP_{main,\{y,z\}}(v)$, i.e., $\neg y(v) \vee \neg t(v)$.

On the other hand, when `s = splice(t, z)` is invoked at $S^c_{3.6}$, the object pointed-to by `y` is a cutpoint. Note, that the formula $isCP_{main,\{t,z\}}(v)$ evaluates to 1 when v is bound to this object: the formula $R_{\{t,z\}}(v)$ holds for every object in `t`'s list. In particular, it holds for the second object which is pointed-to by a local variable, `y`, but not by either one of the actual parameters, `t` or `z`.

Note that $\mathcal{LCPF}$ considers only the values of variables that belong to the current call when it detects cutpoints. This is possible because all pending calls are cutpoint-free.[16]

[16] We note that exploiting the fact that every pending call is guaranteed to be cutpoint-free greatly simplifies the cutpoint detection compared to the method explained in Chapter 4.

Computing The Memory State at the Entry Site

The memory state at the entry site to p (denoted by S_e) represents the local-heap passed to p. It contains only these individuals in S_c that represent objects that are relevant for the invocation. The formal parameters are initialized by $updCall_q^{y=p(x_1,\ldots,x_k)}$, defined in Figure 3.17(a). The latter, specifies the value of the predicates in S_e using predicate-update formulae evaluated over S_c. We use the convention that the updated value of x is denoted by x'. Predicates whose update formula is not specified, are assumed to be unchanged, i.e., $x'(v_1,\ldots) = x(v_1,\ldots)$. Note that only the predicates that represent variable values are modified. In particular, field values, represented by binary predicates, remain in p's local-heap as in S_c.

Example 3.7.3 The structure $S^e_{3.5}$, shown at Figure 3.5, depicts the memory state at the entry-site to `splice` when `t = splice(x, y)` is invoked at the memory state $S^c_{3.5}$. Note that the list referenced by `z` is not passed to `splice`. Also note that the element which was referenced by `x` is now referenced by `p`. This is the result of applying the update formula $p'(v) = x(v)$ for the predicate p in this call. Similarly, the element which was referenced by `y` is now referenced by `q`.

Example 3.7.4 The structure $S^e_{3.6}$, shown at Figure 3.6, depicts the memory state at the entry-site to `splice` when `s = splice(y, z)` is invoked at the memory state $S^c_{3.6}$. Note that the list element pointed to by `x` and `t` is not passed to `splice`. Also note that the element which was referenced by `y` is now referenced by `p`. This is the result of applying the update formula $p'(v) = y(v)$ for the predicate p in this call. Similarly, the element which was referenced by `z` is now referenced by `q`.

Computing The Memory State at the Return Site

The memory state at the return-site (denoted by S_r) is constructed as a combination of the memory state in which p was invoked (denoted by S_c) and the memory state at p's exit-site (denoted by S_x). Informally, S_c provides the information about the (unmodified) irrelevant objects and S_x contributes the information about the destructive updates and allocations made during the invocation.

The main challenge in computing the effect of a procedure is relating the objects at the call-site to the corresponding objects at the return site. The fact that the invocation is cutpoint-free guarantees that the only references into the local-heap are references to parameter objects.[17] This allows us to reflect the effect of p into the local-heap of q by: (i) replacing the relevant objects in S_c with S_x, the local-heap at the exit from p; (ii) redirecting all references to an object referenced by an actual parameter to the object referenced by the corresponding formal parameter in S_x.

We compute the pointer redirection on an intermediate structure, $\langle U', \iota' \rangle$, which contains a copy of $S_c = \langle U_c, \iota_c \rangle$, the memory state at the call site, and of $S_x = \langle U_x, \iota_x \rangle$, the memory state at the exit site.

Every individual in U', the universe of the intermediate structure, "stands" for exactly one individual from either U_c or U_x.

The interpretation function of the intermediate structure, ι', provides meaning for the auxiliary predicates $inUc$ and $inUx$ as well as for every predicate $p \in \mathcal{P}$. The auxiliary predicates are used to distinguish between individuals according to their origin: ι' sets $inUc$ to hold (only) for individuals that originate from U_c. Similarly, it sets $inUx$ to hold (only) for individuals that originate from U_x. For the other predicates, ι' is defined in such a way that it creates disjoint isomorphic[18] copies of S_c and S_x within $\langle U', \iota' \rangle$: Informally, removing from $\langle U', \iota' \rangle$ the individuals that originate from U_c results in a structure which is isomorphic to S_x. Similarly, the removal of the individuals that originate from U_c results in a structure which is isomorphic to S_x.

Pointer redirection is specified by means of predicate update formulae, as defined in Figure 3.17(b). The most interesting aspect of these update-formulae is the formula $match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}$, defined in Figure 3.17. This formula matches an individual that represents an object which is referenced by an actual parameter at the call-site, with the individual that represents the object which is referenced by the corresponding formal parameter at the exit-site. Our assumption that formal parameters are not modified allows us to match these two individuals as representing the same object. Once pointer redirection is complete, the updated memory state of the caller is constructed by removing all the individuals originating from S_c and representing relevant objects of the invocation.

[17] We remind the reader that a parameter object is an object which is pointed to by an actual parameter when the procedure is invoked.

[18] Two structures are isomorphic is they are identical up to renaming of individuals.

Note that while $\langle U', \iota' \rangle$, the intermediate structure, may not be an admissible memory state, the resulting memory state at the return site is admissible.

Example 3.7.5 $S^c_{3.5}$ and $S^x_{3.5}$, shown in Figure 3.5, represent the memory states at the call-site and at the exit-site of the invocation `t = splice(x, y)`, respectively. Their combination according to the procedure call rule is $S^r_{3.5}$, which represents the memory state at the return site. Note that the lists of `x` and `y` from the call-site were replaced by the lists referenced by `p` and `q`. The list referenced by `z` was taken as is from the call-site.

Example 3.7.6 $S^c_{3.6}$ and $S^x_{3.6}$, shown in Figure 3.6, represent the memory states at the call-site and at the exit-site of the invocation `s = splice(y, z)`, respectively. Their combination according to the procedure call rule is $S^r_{3.6}$, which represents the memory state at the return site. Note that the lists of `x` and `y` from the call-site were replaced by the lists referenced by `p` and `q`. The list element pointed to by `x` and `t` was taken from the call-site. The `n`-field from this element to the head of the returned list is created because the list element pointed to by `y` at the call state matches the list element pointed to by `q` at the exit state.

3.7.3 $\mathcal{LCPF}^\sharp$: A Logic-based Abstract Localized-Heap Semantics for Cutpoint-Free Programs

In this section, we present $\mathcal{LCPF}^\sharp$, a conservative abstract semantics abstracting $\mathcal{LCPF}$.

3.7.3.1 Abstract Memory States

We conservatively represent multiple concrete memory states $SS \subset 2Struct$ by a single 3-valued logical structure $S^\sharp \in 3Struct$ using canonical abstraction [SRW02]. (See Definition 2.5.4). Recall that in canonical abstraction each individual from the (concrete) state is mapped into an individual in the abstract state. An abstract memory state may include *summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. (We remind the reader that the Galois connection $(2^{\Sigma_L^{cpf}}, \alpha\colon 2^{\Sigma_L^{cpf}} \rightarrow 2^{3Struct}, \gamma\colon 2^{3Struct} \rightarrow 2^{\Sigma_L^{cpf}}, 2^{3Struct})$ between memory states of $\mathcal{SL}^{\mathrm{CPF}}$ and abstract states, represented using 3-valued logical structures, is obtained by composing β_L^{CPF}, which maps memory states of $\mathcal{SL}^{\mathrm{CPF}}$ to 3-valued logical structures, and canonical abstraction. See Section 3.6.)

Example 3.7.7 Figure 3.5 and Figure 3.6 show the abstract states (as *3-valued* logical structures) representing the concrete states of Figure 3.5 and Figure 3.6, respectively.

Note that only the local variables `p` and `q` are represented inside the call to `splice(p, q)`. Representing only the local variables inside a call ensures that the number of unary predicates to be considered when analyzing the procedure is proportional to the number of its local variables. This reduces the overall complexity of our algorithm to be worst-case doubly-exponential in the maximal number of local variables rather than doubly-exponential in their total number (as in e.g.,[RS01]).

The Importance of Reachability Recording derived properties by means of *instrumentation predicates* may provide additional information that would have been otherwise lost under abstraction. In particular, because canonical abstraction is directed by unary predicates, adding unary instrumentation predicates may further refine the abstraction. This is called the *instrumentation principle* in [SRW02]. In our framework, the predicates that record reachability from variables play a central role. They enable us to identify the individuals representing objects that are reachable from actual parameters. For example, in the *3-valued* logical structure $S^{c\sharp}_{3.5}$ depicted in Figure 3.5, we can detect that the top two lists represent objects that are reachable from the actual parameters because either r_x or r_y holds for these individuals. None of these predicates hold for the individuals at the (irrelevant) list referenced by `z`. We believe that these predicates should be incorporated in any instance of our framework.

3.7.3.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{LCPF^\sharp}{\rightsquigarrow} \subseteq (3Struct \times st) \times 3Struct$. Because our framework is based on [SRW02], the specification of the concrete operational semantics for program statements (as transformers of 2-valued structures) in Section 3.7.2, also defines the corresponding abstract semantics (as transformers of 3-valued structures). This abstract semantics is obtained by reinterpreting logical formulae using a 3-valued logic semantics and serves as the basis for an abstract interpretation. In particular, reinterpreting the side condition of the procedure call rule conservatively, verifies that the *program* is cutpoint-free. In this chapter, we directly utilize the implementation of these ideas available in TVLA [LAS00].

In principle, the effect of a statement on the values of the instrumentation predicates can be evaluated using their defining formulae and the update formulae for the core predicates. In practice, this may lead to imprecise results in the analysis. It is far better to supply the update formula for the instrumentation predicates too. In this chapter, we manually provide the update formulae of the instrumentation predicates (as done e.g., in [SRW02, LARSW00, RS01]). Automatic derivation of update formulae for the instrumentation predicates [RSL03] is currently not implemented in our framework. We note that update formulae are provided at the level of the programming language, and are thus applicable to arbitrary procedures and programs. Predicate update-formulae for the instrumentation predicates are provided in Appendix C.2.1.2.

The soundness of our abstract semantics is guaranteed by the combination of Theorem 3.5.3, and the theorems in [SRW02]

- Theorem 3.5.3 shows that $\mathcal{LSL}^{\text{CPF}}$ is observationally sound with respect to the standard semantics.
- Sagiv et. al. [SRW02] show that every program-analyzer which is an instance of their framework is sound with respect to the concrete semantics it is based on.

3.8 Interprocedural Functional Analysis via Tabulation of Cutpoint-Free Abstract Local-Heaps

In this section, we present a framework for interprocedural shape analysis, which is context- and flow-sensitive with the ability to perform destructive pointer updates. Specifically, we fill the algorithmic details left open in Sections 3.6 and 3.7, which mainly address semantic issues. In particular, we define the effect of call and return statements using first order formulas with transitive closure and show how to compute the effect of program statements.

Our algorithm computes procedure summaries by tabulating input abstract memory-states to output abstract memory-states. The algorithm computes a *partial summary* of every procedure, i.e., the computed tabulation is restricted to abstract memory-states that occur in the *analyzed* program. However, the tabulated abstract memory-states represent local-heaps, and are therefore independent of the context in which a procedure is invoked. As a result, the summary computed for a procedure could be used at different calling contexts and at different call-sites.

Our interprocedural tabulation algorithm is a variant of the IFDS-framework [RHS95] adapted to work with local-heaps. The main difference between our framework and [RHS95] is in the way return statements are handled: In [RHS95], the dataflow facts that reach a return-site come either from the call-site (for information pertaining to local variables) or from the exit-site (for information pertaining to global variables). In our case, the information about the heap is obtained by *combining* pairwise the abstract memory states at the call-site with their counterparts at the exit-site. A detailed description of our tabulation algorithm can be found in Appendix F.

> **Example 3.8.1** Figure 3.18 shows a partial tabulation of abstract local-heaps for the `splice` procedure of the running example. As we have already mentioned, the splice procedure is only analyzed 9 times before its tabulation is complete, producing a summary that is then reused whenever the effect of `splice(p, q)` is needed. The figure shows all 9 possible input states.

3.8.1 Prototype Implementation

We have implemented a prototype of our framework using TVLA [LAS00]. The framework is parametric in the heap-abstraction and in the operational semantics. We have instantiated the framework to produce a shape-analysis algorithm for analyzing Java programs that manipulate (sorted) singly-linked lists and unshared trees. To

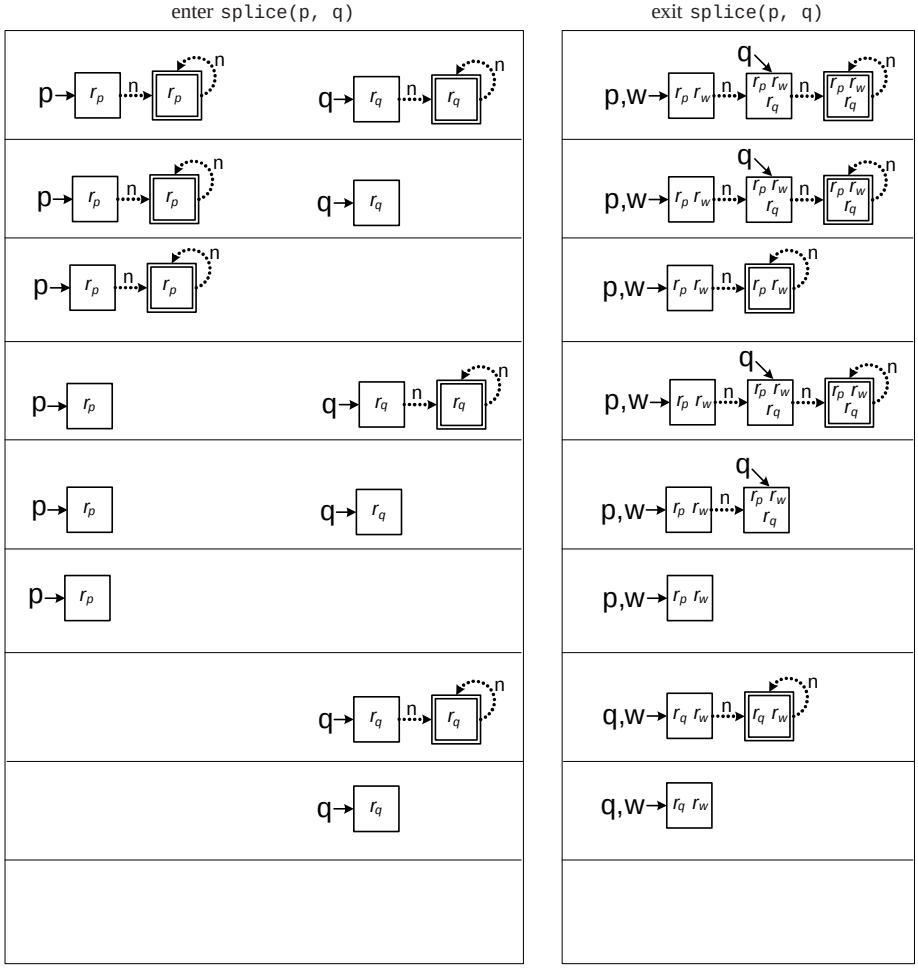


Figure 3.18: The tabulation of abstract states for the `splice` procedure as computed by our analysis in the running example. (In the seventh row, eighth row, and ninth row, the value of the formal parameter p is *null*, and thus it is not shown. For similar reasons, q is not shown in the third row, sixth row, and ninth row. Specifically, in the ninth row, when the both arguments to `splice` have a *null* value, the return value of `splice` is also *null*.) We note that in this program, the analysis actually computed a complete tabulation for (cutpoint-free invocations of) `splice`.

Iterative vs. Recursive Programs				
Implementation	**Iterative**		**Recursive**	
a. List manipulating programs	**Space**	**Time**	**Space**	**Time**
create creates a list	2.5	11.5	2.3	9.3
find searches an element in a list	3.2	23.7	3.6	37.1
insert inserts an element into a sorted list	5.1	50.1	5.4	46.8
delete removes an element from a sorted list	3.7	41.7	3.9	35.8
append appends two lists	3.7	18.4	3.9	22.5
reverse destructive list-reversal	3.6	26.9	3.4	21.0
revApp reverses a list by appending its head to its reversed tail	4.3	43.6	4.3	41.7
merge merges two sorted lists	12.5	585.1	5.4	87.1
splice splices two lists	4.9	76.5	4.8	33.6
running the running example	5.2	80.5	5.0	36.5
b. Tree manipulating programs	**Space**	**Time**	**Space**	**Time**
create creates a full tree	-	-	2.6	14.3
insert inserts a node	5.4	98.1	5.6	49.6
remove removes a node using `removeRoot` and `spliceLeft`	9.6	480.3	6.6	167.5
find finds a node with a given key	4.9	53.4	6.5	105.7
height returns the tree's height	-	-	5.4	76.1
spliceLeft a tree as the leftmost child of another tree	5.3	51.6	5.3	35.7
removeRoot removes the root of a tree	6.1	107.8	6.1	73.9
rotate rotates the left and right children of every node	-	-	4.9	57.1

c. [RS01] (Call String) vs. [JLRS04] (Relational) vs. our method						
Method	**Call String**		**Relational**		**Our method**	
Procedure	**Space**	**Time**	**Space**	**Time**	**Space**	**Time**
insert	1.8	20.8	6.3	122.9	3.5	20.0
delete	1.7	16.4	6.8	145.7	2.8	14.9
reverse	1.8	13.9	4.0	6.4	2.8	7.5
reverse8	2.7	123.8	9.1	14.8	2.8	21.7

d. Inline vs. Procedural Abstraction				
	Inline		**Proc. Call**	
Program	**Space**	**Time**	**Space**	**Time**
crt1x3	2.5	5.1	2.5	6.0
crt2x3	4.5	12.5	2.8	7.3
crt3x3	6.4	22.6	3.1	8.6
crt4x3	8.1	38.6	3.3	9.9
crt8x3	17.3	133.4	4.0	15.6

Table 3.1: Experimental results. Time is measured in seconds. Space is measured in megabytes. Experiments performed on a machine with a 1.5 Ghz Pentium M processor and 1 Gb memory.

translate Java programs we have extended an existing Soot-based [VRCG+99] front-end for Java developed by R. Manevich.

The join operator in our framework can be either set-union or a more "aggressive" partial-join operation [MSRF04]. The former ensures that the analysis is fully-context sensitive. The latter exploits the fact that our abstract domain has a Hoare order and returns an upper approximation of the set-union operator.[19] Our experiments were conducted with the partial-join operator.

Our analysis was able to verify that all the tested programs are cutpoint-free and *clean*, i.e., do not perform null-dereference and do not leak memory. For singly-linked-list-manipulating programs (Table 3.1.a), we also verified that the invoked procedures preserve list acyclicity. The analysis of the tree-manipulating programs (Table 3.1.b) verified that the tree invariants hold after the procedure terminates. For these programs we assume (and verify) that the trees are unshared.

[19] Informally, the partial join operator merges together 3-valued logical structures according to a partial isomorphism similarity criteria. The resulting structure conservatively represents each of the merged structure. The operator considers two 3-valued structures $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$ to be partially isomorphic (and thus merges them) when (i) $\iota(p) = \iota'(p)$ for all predicates $p \in \mathcal{P}$ with arity 0, and (ii) for every individual $u \in U$ there exists an individual $u' \in U'$ such that $\iota(p)(u) = \iota(p)(u')$ for every unary predicate $p \in \mathcal{P}$, and vice versa. Predicates of arity 2 or more are given a definite value (0 or 1) only when it is consistent with all the merged structures, i.e., when the resulting structure conservatively represents each of the merged structures, and an indefinite value ($1/2$) otherwise. For further details, see [MSRF04]. The information loss incurs by the partial-join operator can be formalized as the most abstract stage in the approximation hierarchy of abstract interpretations which is used to derive our analysis. See, e.g., [Cou97].

Procedure	Space (MB)	Time (sec)
createLayer1	2.1	3.6
createLayer2	2.5	7.4
createLayer4	2.9	15.5
createLayer8	3.8	34.7
createLayer16	5.5	83.6
createLayer32	9.0	230.0
createLayer64	15.5	664.9

Procedure	Space (MB)	Time (sec)
NTypesAltenate1	2.0	2.5
NTypesAltenate2	2.5	8.2
NTypesAltenate4	3.3	21.9
NTypesAltenate8	5.4	61.6
NTypesAltenate16	10.5	232.1

Table 3.2: Cost of the analysis for programs that invokes procedures in many irrelevant contexts.

3.8.1.1 Cost of Analysis

Table 3.1a-b and Table 3.3a-b compare the cost of analysis for iterative and recursive implementations of a given program.[20] For these programs, we found that the cost of analyzing recursive procedures and iterative procedures is comparable in most cases. We note that our tests were of *client* programs and not a single procedure, i.e., in all tests, the program also allocates the data structure that it manipulates.

Table 3.1.c shows that our approach compares favorably with existing TVLA-based interprocedural shape analyzers [RS01, JLRS04]. The experiments measure the cost of analyzing 4 recursive procedures that manipulate singly linked lists. For fair comparison with [RS01] and [JLRS04], we follow them and do not measure the cost of list allocation in these tests. All analyzers successfully verified that these (correct) procedures are clean and preserve list acyclicity. [JLRS04] was able to prove that `reverse` reverses the list and to pinpoint the location in the list that `delete` removed an element from. However, the cost of analysis for `insert` and `delete` in [JLRS04] was higher than the cost in [RS01] and in our analysis. Procedure `reverse8` reverses the same list 8 times. The cost of its analysis indicates that our approach, as well as [JLRS04], profits from being able to reuse the summary of `reverse`, while [RS01] cannot.

In addition, we examined whether our analysis benefits from reuse of procedure summaries. Table 3.1.d shows the cost of the analysis of programs that allocate several lists. Program **crtYx3** allocates Y lists. The table compares the cost of the analysis of programs that allocate a list by invoking `create3` (right column) to that of programs that inline `create3`'s body. The results are encouraging as they indicate (at least in these simple examples) that our analysis benefits from procedural abstraction.

Table 3.2 demonstrates how our approach benefits from ignoring irrelevant parts of the heap. Procedure `createLayerY` creates a full binary tree of depth Y. It operates this by first creating two full trees of depth $X = Y - 1$ by invoking the procedure `createLayerX` twice; and then it sets these trees as the children of a tree node that it allocates. Because the `createLayer` procedures are parameterless, none of the objects that are allocated when it is invoked is relevant. In particular, the local-heap at the entry to the procedure is always empty, thus we analyze the body of every one of these procedures only once. Using the method [JLRS04], we would have to analyze every procedure at least twice: one time when it is invoked when the rest of the heap is empty and once when it is invoked when the rest of the heap contains some objects. Using the method [RS01], we would have to analyze procedure `createLayerX` at least $Z - X$ times when allocating a tree of depth Z because the analysis distinguishes between memory states in which the pending activation records already have a reference to a tree or not. The method of [CR03] does not handle trees.

Procedure `NTypesAltenateY` allocates a list with Y elements, where the i-th element is of class `NTypesI`. This procedure allocates the list by non deterministically choosing whether to first allocate the head of the list first and then invoking `NTypesAltenateX` (for $X = Y-1$) to allocate the list's tail or vice versa. Again, because these procedures are parameterless our method analyzes every procedure only once. However, the methods of [JLRS04] and [RS01] will analyze procedure `createLayerX` at least 2^{Z-X} times when allocating a list of length Z, once for every possible combination of objects allocated by the pending calls. We believe that the method of [CR03] would also analyze these procedures only once.

[20] `revApp` is a recursive procedure. We analyzed it once with an iterative append procedure and once with a recursive append. Tail sort is a recursive procedure. We analyzed it once with an iterative insert procedure and once with a recursive insert.

Implementation	Iterative		Recursive	
a. Sorted list manipulating programs	**Space**	**Time**	**Space**	**Time**
create creates a list	2.6	15.0	2.4	12.3
find searches an element in a list	4.1	42.2	4.6	60.7
insert inserts an element into a sorted list	4.7	67.7	4.9	69.5
delete removes an element from a sorted list	4.7	75.1	4.8	70.4
reverse destructively reverses a sorted list	4.5	54.8	4.4	47.6
revApp reverses a sorted list by appending its head to the reversed tail	4.8	74.8	4.8	70.5
merge merges two sorted lists	11.7	1115.1	5.8	131.2
last returns the last element in a sorted list	4.2	43.3	4.5	47.3
maxFirst returns the first maximal element in an unsorted list	3.6	71.3	3.5	51.6
maxLast returns the last maximal element in an unsorted list	3.6	83.5	3.5	59.0
minFirst returns the first minimal element in an unsorted list	3.5	64.1	3.5	51.1
minLast returns the last minimal element in an unsorted list	3.9	76.3	3.5	59.0
b. Sorting programs	**Space**	**Time**	**Space**	**Time**
IinsertionSort moves the list elements into a sorted list	8.6	449.8	7.3	392.2
TailSort inserts the list head to its (recursively) sorted tail	4.9	101.6	4.9	103.4
QuickSort quicksorts a list	-	-	13.5	1017.1

Table 3.3: Experimental results for additional sorting programs. Time is measured in seconds. Space is measured in megabytes. Experiments performed on a machine with a 1.5 Ghz Pentium M processor and 1 Gb memory.

3.8.1.2 Analyzing Sorting Programs

The analysis presented in [LARSW00] allows to prove partial correctness of sorting and list manipulating procedures. The main idea [LARSW00] is to track the relative order between the data components of list elements using a binary core relation. Section C.3 describes our adaptation of their abstraction to local-heaps. It also provides a rather detailed description of the analysis of `quicksort`. (We note that prior attempts to verify the partial correctness of `quicksort` using TVLA were not successful.)

We applied our analysis to verify the iterative and recursive sorting programs listed in Table 3.3. Our analysis was able to verify that these programs are clean and preserve list acyclicity. Furthermore, it verified that:

(i) `find`, `last`, `insert`, and `delete` preserve list sorted-ness.
(ii) `merge` merges two sorted lists into one sorted list.
(iii) Reversing a sorted list by either `reverse` or `revApp`, results in a list in reversed order.
(iv) `MaxFirst` returns the list element with the highest data value in the list, and that its value is *strictly* greater than that of any preceding element. Similar properties were verified for the procedures `MaxLast`, `MinFirst`, and `MinLast`.
(v) The sorting programs (Table 3.1.c) return a sorted permutation of their input.

For further details pertaining to the analysis of sorting programs, and in particular of the analysis of `quicksort`, see Section C.3.

3.8.1.3 Dead Cutpoints

For two of our example programs (`quicksort` and `reverse8`), cutpoints were created as a result of objects pointed-to by a dead variable or a dead field at the point of a call. We manually rewrote these programs to eliminate these dead references, thus making the programs cutpoint-free.

Chapter 4

Interprocedural local-heap Shape Analysis for Programs with Cutpoints

This chapter presents an approach to the interprocedural local-heap shape analysis of imperative languages with procedures and dynamically allocated storage. It develops shape analysis algorithms which are context- and flow-sensitive with the ability to perform destructive pointer updates. The developed algorithms are abstract interpretations of $\mathcal{LSL}$, a novel non-standard storeless *semantics which is shown to be* observationally equivalent *to the standard semantics of heap manipulating programs.*

The distinguishing aspect of $\mathcal{LSL}$, and of its abstractions, is that they compute a characterization of a procedure's behavior which abstracts away the contents *of the parts of the heap which are irrelevant to the procedure: Only the contents of the procedure's local-heap and its* (arbitrary) sharing patterns with the rest of the heap *are recorded.*

The material described in this chapter is largely based on the material that originally appeared in [RBR$^+$05, RBR$^+$04, RSY04].

4.1 Introduction

This chapter presents a framework for interprocedural shape analysis, which is context- and flow-sensitive with the ability to perform destructive pointer updates. In this chapter, we do not place an *a priori* restriction on the class of programs that can be effectively analyzed. Specifically, we are interested in analyzing programs with cutpoints (cf. the interprocedural framework presented in Chapter 3, in which we restricted our attention *a priori* to the specific class of cutpoint-free programs). In particular, the analysis algorithms developed in this chapter are capable of conservatively representing arbitrary sharing patterns between the part of the heap which is relevant to the procedure and the other, irrelevant, parts.

Technically, our analysis computes procedure summaries as transformers from inputs to outputs while *recording only the sharing patterns with parts of the heap not relevant to the procedure and abstracting away the contents of the irrelevant parts*. This makes the analysis modular in the heap and thus allows reusing the effect of a procedure at different contexts in which the irrelevant parts of the heap induce *similar sharing patterns*.

This chapter consists of two main parts: The first part introduces a non-standard concrete semantics, $\mathcal{LSL}$, for *Localized-heap Store-Less*. In $\mathcal{LSL}$, called procedures are only passed *parts* of the heap. $\mathcal{LSL}$ *can handle programs with arbitrary sharing patterns between the local-heap and the irrelevant heap context* by providing a special treatment for cutpoints. $\mathcal{LSL}$ is shown to be *observationally equivalent* to a standard store-based semantics. The second part of this chapter concerns abstract interpretation of $\mathcal{LSL}$ and develops new static-analysis algorithms using canonical abstraction [SRW02]. Specifically, our analysis allows for a parametric abstraction of the sharing patterns.

4.1.1 $\mathcal{LSL}$: A Localized-Heap Storeless Semantics

In Section 4.4, we introduce $\mathcal{LSL}$, a non-standard concrete semantics. $\mathcal{LSL}$ is a *storeless* semantics [Jon81]: Every dynamically allocated object o is represented by the set of *access paths* that reach o. In particular, unreachable objects are not represented. $\mathcal{LSL}$ is a *local-heap* semantics: It does not represent access paths that start from variables of pending calls in the "local state" of the current procedure. This means that a procedure has a local view that only includes objects that are reachable from the procedure's parameters (and, in addition, any objects that it allocates). In this aspect, $\mathcal{LSL}$ is similar to $\mathcal{LSL}^{\text{CPF}}$ and differs from existing storeless semantics for procedural programs [Deu92a, Ven99], which explicitly represent the global-heap and, in particular, represent the values of pending variables.

In contrast to the $\mathcal{LSL}^{\text{CPF}}$ semantics, $\mathcal{LSL}$ allows cutpoints. The main challenge in handling procedure calls with cutpoints in a storeless semantics is that the "same object" can be represented differently at different program states. In particular, *cutpoints*, the objects which separate the "local-heap" that can be accessed by a procedure from the rest of the heap, can be represented differently at the entry state and at the exit state.

Our main observation in the design of $\mathcal{LSL}$ is that *for the purpose of applying the effect of a procedure call*, it suffices to *only* match the representation of the parameter objects (i.e., the objects pointed to by the formal parameters) and the *cutpoints* of the invocation at the entry state and at the exit state. In particular, there is no need to match the representation of *every* object at the entry state with its representation at the exit state.

Technically, $\mathcal{LSL}$ addresses the problem of relating objects at the caller's call state with objects at the callee's exit state in a storeless semantics in two complementary ways:

- $\mathcal{LSL}$ matches the (possibly different) representation of the parameter objects using the (immutable) values of the formal parameters.[1] Thus, like in $\mathcal{LSL}^{\text{CPF}}$, a formal parameter serves as an immutable marker which keeps labeling the same (parameter) object throughout the execution of the procedure.
- $\mathcal{LSL}$ matches the (possibly different) representation of the parameter objects using *cutpoint-labels*: $\mathcal{LSL}$ labels every cutpoint with a *canonic* immutable *cutpoint-label*, which is incorporated in the representation of the object in the memory state.

The main technical novelty of $\mathcal{LSL}$ is the use of *cutpoint-labels*. Technically, a cutpoint-label is the set of access paths that point to the (cutpoint) object at the entry to the procedure. As a result, cutpoint-labels allow to record the sharing patterns between the procedure's local-heap and the rest of the heap in a *canonic* and context-independent manner. Specifically, cutpoint-labels address the problem of relating properties before and after a call in a storeless semantics by labeling every cutpoint with a *canonic* immutable *cutpoint-label*; when a procedure returns, the cutpoint-labels are used to match the cutpoint object of the invocation at the entry state and at the exit state. This allows to update the caller's local-heap with the effect of the call. (See Section 4.3)

We study the properties of $\mathcal{LSL}$: We show that $\mathcal{LSL}$ is observationally *equivalent* with a standard store-based semantics. We also show that it has a number of standard properties, including full abstraction and determinism. (See Section 4.5).

4.1.2 Interprocedural Shape Analysis

We develop interprocedural shape analysis algorithms by abstract interpretation of $\mathcal{LSL}$. The latter was designed with its precise and efficient abstractions in mind: information about the context provided by the rest of the heap is isolated to the sharing patterns of the cutpoints—which are expressible in a context-independent manner. The analysis benefits from the fact that the heap is localized: the behavior of a procedure only depends on the part of the heap that is reachable from actual parameters, and on the sharing patterns that create cutpoints. Furthermore, because our analyses exploit the aforementioned beneficial aspects of $\mathcal{LSL}$, their results can be reused for different contexts that have similar sharing patterns.

The interprocedural algorithms developed in this chapter follow the functional approach [CC78, SP81] (see Section 6.3.1.2): They tabulate abstractions of memory configurations before and after procedure calls. However, these abstractions are represented in a non-standard way *abstracting away parts of the heap not relevant to the procedure*. This reduces the complexity of the analysis because the analysis of procedures does not represent information on references and the heap from calling contexts. Indeed, this makes the analysis modular in the heap context (*heap modular*), and thus allows reusing the effect of a procedure at different calling contexts.

[1] Recall that by our simplifying assumptions, formal parameters are never assigned. (See Section 2.2.1).

One of the most subtle issues in our analysis is the treatment of sharing between the local-heap and the rest of the heap. The problem is that, unlike in cutpoint-free programs, here the local-heap can be accessed via access paths which bypass actual parameters. Therefore, cutpoints are treated differently than other objects: Our analyses allow the specifier to define the expected sharing patterns. (We note that the analysis may become imprecise, and expensive, in programs in which several cutpoints are summarized together).

Indeed, it is instructive to distinguish between two dimensions of heap abstractions: (i) The abstraction of the local-heap which discriminates between different kinds of aliases inside the part of the reachable part of the heap. For example, a node which is pointed-to by two or more selectors from the local-heap can be treated differently. (ii) The abstraction of the *sharing patterns* between the local-heap and the rest of the heap. For example, in some of our experiments our analyses lose all distinctions between the different cutpoints. In other experiments, the analyses distinguish only between cutpoints with different types. We note that using these schemes leads to a loss of precision when more than one cutpoint (of the same type) is created. (See Section 4.10.4.1).

We develop *conservative* algorithms: they compute a conservative description of every memory state that can arise (at any program point) in any execution. This means that we can conservatively determine properties of the program such as the absence of null-dereferences, absence of garbage, preservation of data-structure invariants, and validity of assertions by checking these properties on the (generated) abstract states. However, because the description is *conservative*, the algorithms might represent concrete states that are infeasible according to the concrete semantics. This leads to incompleteness in the sense that we may fail to establish assertions that hold for every execution.

Technically, iterative abstract-interpretation algorithms of $\mathcal{LSL}$ are used to automatically compute a safe approximation to the set of possible program states. The main idea is that every abstract state finitely represents a potentially infinite number of concrete $\mathcal{LSL}$ states. The program is interpreted according to an abstract semantics that over approximates the concrete transition relation. Termination of the abstract-interpretation algorithms is guaranteed by the finiteness of the set of abstract states.

Our analyses extend the 3-valued logical framework for program analysis of [LAS00, SRW02] to the interprocedural setting. Furthermore, they leverage the parametric nature of this framework and are parametric in the heap abstraction and in the concrete effects of program statements, allowing to experiment with different instances of interprocedural shape analyzers. For example, we can employ different abstractions for singly-, doubly-linked lists, and trees. (See Section 4.9.1).

Perhaps the main technical challenge in encoding local-heaps with cutpoints using the framework of [LAS00, SRW02] is the (conservative) representation of the *access paths comprising* cutpoint-labels. We address this issue by "freezing" the entry memory states (whose structure defines the cutpoints) in the current memory state of the procedure. Effectively, our analysis abstracts every memory state that occurs during the execution of a procedure *together* with the memory state that occur at the entry to the procedure. The "frozen" entry memory state is used to encode the cutpoint-labels.

The soundness of our algorithms is guaranteed by a combination of the theorems given in this chapter and the ones given in [SRW02]. (See Section 4.8.3).

4.1.3 Main Results

The main contributions presented in this chapter can be summarized as follows:

- We introduce the novel notion of *cutpoint-labels*, which allow to record the sharing patterns between the procedure's local-heap and the rest of the heap in a context-independent manner.

- We develop $\mathcal{LSL}$, a non-standard concrete procedure local-heap *storeless* semantics. $\mathcal{LSL}$ utilizes cutpoint-labels to handle procedure calls with arbitrary cutpoints. We show that $\mathcal{LSL}$ is *observationally equivalent* to a standard store-based semantics.[2]

- We present a framework for interprocedural shape analysis that does not impose an a priori restriction on the allowed sharing patterns. In particular, the framework described in this section can handle programs that are not cutpoint-free.

[2] Compare, in contrast, $\mathcal{LSL}^{\text{CPF}}$ which is *observationally sound* with respect to a standard store-based semantics.

- We implemented our framework and used it to prove the absence of program errors such as null dereferences and memory leaks, and to verify conformance to API specifications in several small heap-intensive Java programs.

Remark 4.1.1 *We note that abstract interpretation of $\mathcal{LSL}$ can also provide insights into Deutsch's work on static may-alias analyses based on pointer-access paths [Deu94]. In particular, a may-alias abstraction of $\mathcal{LSL}$ provides insights into the treatment of variables of pending calls, which is one of the most complicated aspects of [Deu94]. For instance, a surprising aspect of the method given in [Deu94] is that recursive procedures are handled in a more precise way than loops. The intuitive reason is that the abstraction of values of variables in the current procedure differs from the abstraction used for values of variables in pending procedures. It is possible to understand the technical reasons for this behavior of the analysis by formalizing the abstract domain of [Deu94] as an abstraction of (the power domain of) $\mathcal{LSL}$ memory states. (See Section D.4).*

Outline. The remainder of the chapter is organized as follows: Section 4.2 introduces our running example. Section 4.3 defines the notion of cutpoint-labels and describes their use in $\mathcal{LSL}$. Section 4.4 presents the $\mathcal{LSL}$ semantics and Section 4.5 investigates its properties. Sections 4.7–4.9 present our interprocedural shape analysis: Section 4.6 presents an informal overview of the analysis. Section 4.7 describes a shape abstraction of $\mathcal{LSL}$, Section 4.8 describes the abstract transformers, and Section 4.9 presents our implementation and experimental evaluation.

4.2 Motivating Example

In this chapter, we continue using as our running example the two programs using procedure `splice`, shown in Figure 3.1, which also served as the running example of Chapter 3. For each invocation of `splice` in these programs, our analyzer verifies that the returned list is acyclic and not heap-shared;[3] that the first parameter is aliased with the returned reference; and that the second parameter points to the second element in the returned list.

> **Example 4.2.1** Figure 4.1(a) depicts four memory states that may arise during the invocation `t = splice(x, y);` according to the $\mathcal{LSB}$ semantics (see Section 2.3) using the same graphical conventions introduced in Example 2.2.1. Figure 4.1($s_L^{c\,4.1}$) depicts the call memory state; Figure 4.1($s_L^{e\,4.1}$) depicts the entry memory state; Figure 4.1($s_L^{x\,4.1}$) depicts the exit memory state; and Figure 4.1($s_L^{r\,4.1}$) depicts the return memory state.
>
> Recall that in $\mathcal{LSB}$, the heap at the entry state is a restriction of the heap at the call state on the set of relevant objects for the invocation. For this invocation of `splice`, only the elements in `x`'s list and in `y`'s list are relevant. Thus, the entry state of this invocation ($s_L^{e\,4.1}$) does not represent the elements in `z`'s list.
>
> The heap of the return state is comprised of `splice`'s heap at the exit state combined with the parts of the heap that were irrelevant for the call. Specifically, it contains `z`'s list. The environment at the return state is as in the call-site, except that the return value is assigned to `t`.

Note that the invocation in Example 4.2.1 is cutpoint-free. In this chapter, we concentrate on the non cutpoint-free program shown in Figure 3.1(b), and in particular on the invocation `s = splice(t, z)`, which is *not* cutpoint-free.

> **Example 4.2.2** Figure 4.2(a) depicts four memory states that may arise during the invocation `s = splice(t, z);` according to the $\mathcal{LSB}$ semantics (see Section 2.3) using the same graphical conventions introduced in Example 2.2.1. Figure 4.2($s_L^{c\,4.2}$) depicts the call memory state; Figure 4.2($s_L^{e\,4.2}$) depicts the entry memory state; Figure 4.2($s_L^{x\,4.2}$) depicts the exit memory state; and Figure 4.2($s_L^{r\,4.2}$) depicts the return memory state.
>
> Recall that in $\mathcal{LSB}$, the heap at the entry state is a restriction of the heap at the call state on the set of relevant objects for the invocation. In this invocation, all the allocated objects are relevant for the

[3] An object is heap-shared if it is pointed-to by a field of more than one object.

invocation. Thus, in this invocation `splice`'s local-heap at the entry state is identical to `main`'s local-heap at the call state. (Note that, in this invocation, `splice`'s local-heap contains all the allocated objects.)

The heap of the return state is comprised of `splice`'s heap at the exit state combined with the (empty) irrelevant parts of the heap at the call state. The environment at the return state is as in the call-site, except that the return value is assigned to `s`.

Note that at the call state, `y` points to the second list element in `t`'s list. Thus, this element is a cutpoint. As expected, `y` points to the same list element and the return state. The semantics can have `y` maintain in the return state the value that it had at the call state because the location (i.e., address) identifying an object is immutable (see Section 2.3). Note, however, that after this invocation returns, the tail of the list pointed-to by `y` has changed.

4.3 Cutpoints and Cutpoint-Labels

In this section, we define the notion of cutpoint-labels and describe their use in $\mathcal{LSL}$. To assist the reader, we provide some intuition by referring to the global store-based semantics (see Section 2.2) and to a small-step [Plo81, NNH99] stack-based operational semantics. $\mathcal{LSL}$ is a storeless semantics, i.e., memory cells are not identified by locations. Thus, we cannot talk about locations as in Sections 2.2 and 2.3. Instead, as in Chapter 3, we use the term *objects*.

In $\mathcal{LSL}$, every dynamically allocated object o is represented by the set of pointer-access paths that reach o. Like $\mathcal{LSL}^{\text{CPF}}$, and unlike other storeless semantics, e.g., [Deu92a], in $\mathcal{LSL}$, pending access paths are not explicitly represented as parts of the local state of the current procedure.

The advantage of our approach is that when a procedure is invoked, it operates only on a part of the heap, namely, the objects that are reachable from the procedure's actual parameters. This allows analyses that abstract $\mathcal{LSL}$ to capture procedural abstraction in a context independent way by inferring the effect of the procedure as a transformer of parts of heaps and not as a transformer of whole heaps.

The downside of this approach is that the memory state just after the call cannot always be defined in terms of the state prior to the call. The intuitive reason for this deficiency is that the description of an object may change due to destructive updates. For example, in the running example, to determine that the pointer-access paths `y` and `x.n.n` are aliased after the second invocation of `splice` in Figure 3.1(b), we need to know that the list element pointed-to by `p.n` when the execution of `splice` begins, is pointed-to by `w.n.n` when the execution ends. To capture this kind of temporal relationship, $\mathcal{LSL}$ tracks the effect of a procedure on *cutpoints* (see Definition 3.3.2).

4.3.1 Cutpoint-Labels

Technically, $\mathcal{LSL}$ uses *cutpoint-labels* to relate the post-state of the procedure with its pre-state. Cutpoint-labels mark the cutpoints at—and throughout—an invocation.

Definition 4.3.1 (Cutpoint-Labels) *A **cutpoint-label*** $\text{cpl} \in 2^{F_p \times \Delta}$ *for procedure p is a set of access paths that start at a formal parameter of p. The set $2^{F_p \times \Delta}$ is denoted by $CPLbs_p$.*

In every procedure invocation, $\mathcal{LSL}$ labels all the cutpoints. A cutpoint-label is the set of all access paths that start with a formal parameter (of the invoked procedure) and point-to the cutpoint when the procedure execution starts. The label of a cutpoint does not change throughout the execution of the procedure's body, even if the heap is modified by destructive updates.

For example, the second list element in `x`'s list is a cutpoint for the invocation `s = splice(t, z);`. The label of this cutpoint is $\{p.n\}$ because $p.n$ is the (only) access path that points-to the cutpoint at the entry to the procedure. A good analogy for the role of cutpoint-labels in our semantics is the use of auxiliary variables in formal verification. Auxiliary variables are used to record variable values at the entry to a procedure; a cutpoint-label is used to record the access paths that reach a cutpoint at procedure entry. To emphasize this similarity, we use the notation $\widehat{a}$ where $a \in CPLbs_p$ for cutpoint-labels for procedure p.

$\mathcal{LSL}$ can infer the effect of an invoked procedure on the heap of its caller by including in the representation of an object all the field paths that reach it and start at a cutpoint.

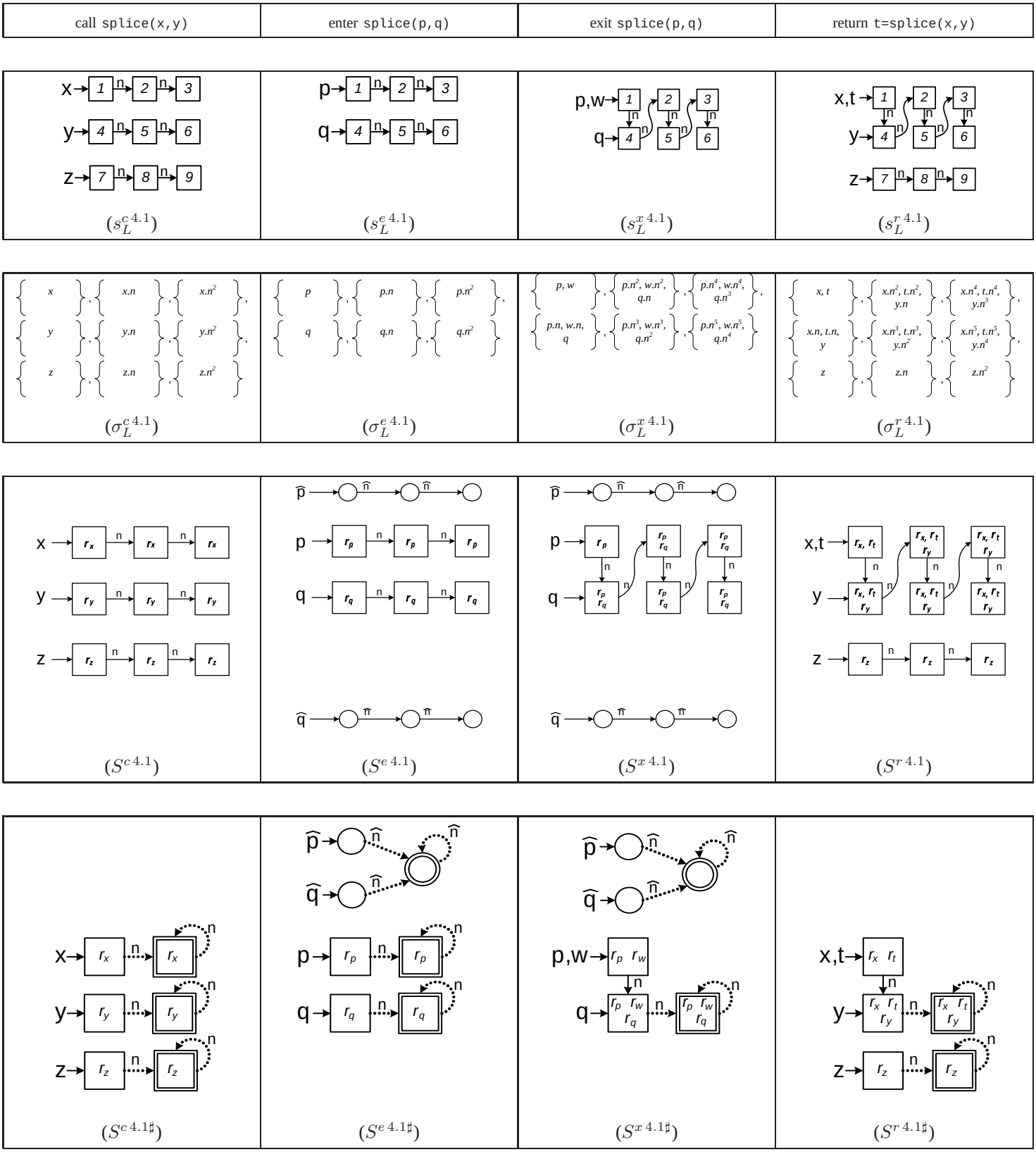


Figure 4.1: Concrete states for the invocation `t = splice(x, y)` in the running example according to the (a) $\mathcal{LSB}$ semantics, (b) $\mathcal{LSL}$ semantics, (c) $\mathcal{LCP}$ semantics, and (d) $\mathcal{LCP}^\sharp$ semantics. We use $x.n^k$ as a shorthand for an access path rooted at x and traversing k n-fields. For example, we use $x.n^3$ as a shorthand for $x.n.n.n$.

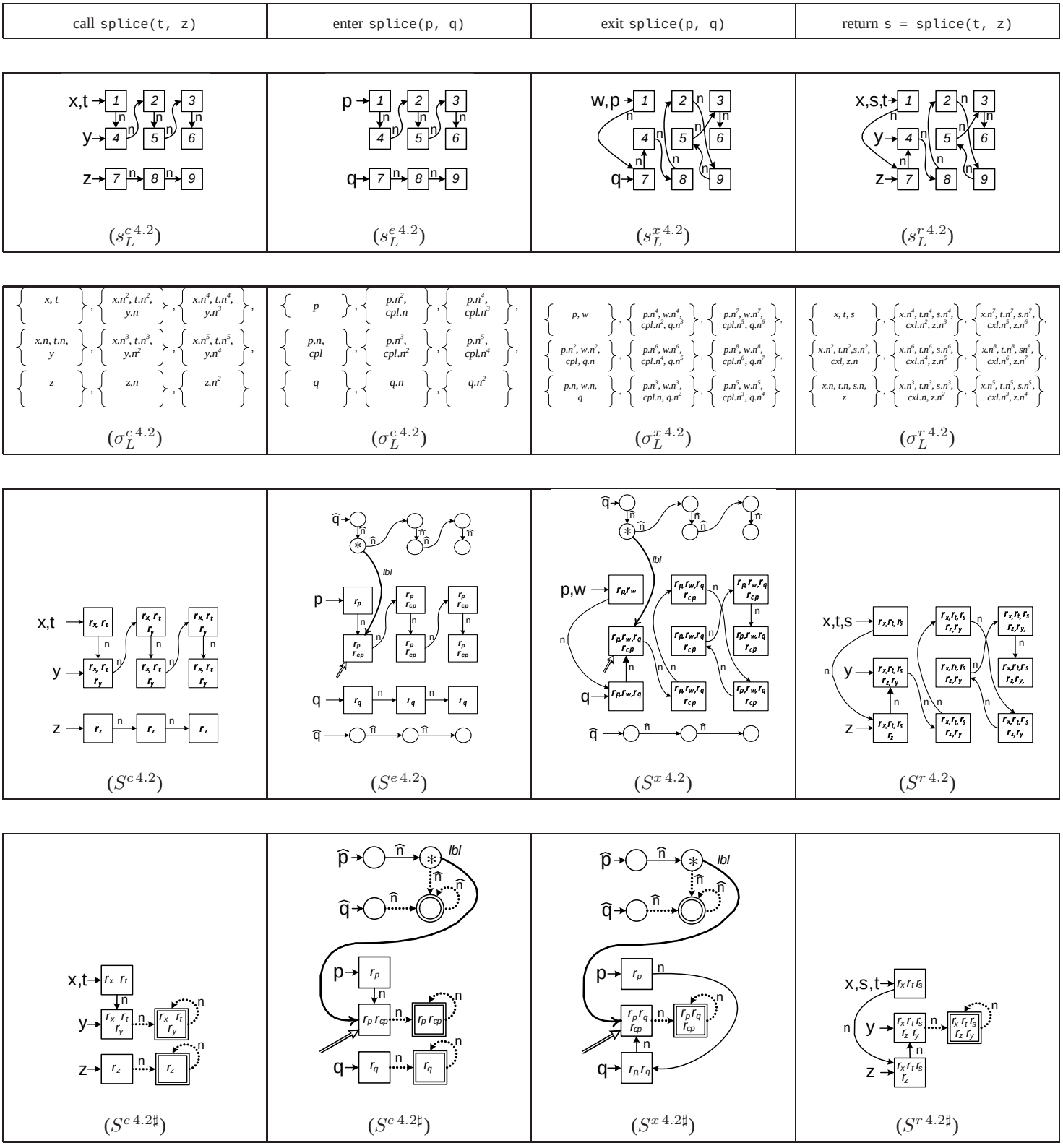


Figure 4.2: Concrete states for the invocation `s = splice(t, z)` in the running example according to the (a) $\mathcal{LSB}$ semantics, (b) $\mathcal{LSL}$ semantics, (c) $\mathcal{LCP}$ semantics, and (d) $\mathcal{LCP}^\sharp$ semantics. We use $x.n^k$ as a shorthand for an access path rooted at x and traversing k n-fields. For example, we use $x.n^3$ as a shorthand for $x.n.n.n$. Also, *cpl* is a shorthand for $\widehat{\{p.n\}}$.

Definition 4.3.2 (Cutpoint-anchored paths) *A **cutpoint-anchored path** $\alpha = \langle \text{cpl}, \delta \rangle \in CPLbs_p \times \Delta$ for a procedure p is a cutpoint-label for procedure p and a (possibly empty) sequence of fields.*

For example, at the exit memory state of the invocation `s = splice(t, z);`, the cutpoint-anchored path $\widehat{\{p.n\}}$ is aliased with the access path $w.n.n$. From this information, our semantics can infer that in the `main` procedure, at the state after the invocation of `splice`, the caller's local variable y is aliased with $t.n.n$.

Technically, during the invocation of a procedure, an object is represented by the access paths and cutpoint-anchored paths that point-to it.

Definition 4.3.3 (Generalized access paths) *A **generalized access path** for a procedure p is either an access path of p or a cutpoint-anchored path of p. $GAccPath_p$ denotes the set of all access paths of procedure p. $GAccPath$ denotes the union of all access paths of all procedures in a program.*

When there is no risk of confusion, we abbreviate a generalized access path of the form $\langle r, \epsilon \rangle$ by r. Note that r can be either a variable, or a cutpoint-label.

Remark 4.3.4 *Cutpoint-labels isolate the information about the part of the heap that a procedure cannot access, to the* sharing pattern *of the cutpoints, i.e., to the set of access paths that—at the entry to the procedure—point-to a cutpoint. Furthermore, the isolation is achieved in a parametric way: although a cutpoint-label expresses the fact that an object is also pointed-to by a pending access path, it is described in terms of the invoked procedure's formal parameters. This allows us to infer the meaning of a cutpoint-label in a context-independent way.*

Remark 4.3.5 *Note that because of the "garbage-collecting nature" of storeless semantics, there is a non-trivial technical difficulty in obtaining a local semantics for the storeless model. If a garbage-collection scan was to collect the heap using only the procedure's local variables as the roots, then elements would be garbage collected that are accessible in the global state; adding the cutpoint-labels to the set of "roots" prevents this potential source of unsoundness. In theory, this idea can be used to develop different garbage collection algorithms for sequential programs.*

4.4 $\mathcal{LSL}$: A Localized-Heap Storeless Semantics

In this section, we define $\mathcal{LSL}$, a Localized-heap Store-Less semantics. The semantics is a natural semantics and, as before, tracks only pointer values.

To define the semantics, we extend the infix operator $\cdot\cdot$, defined in Figure 3.7 and explained in Section 3.4, to generalized access paths in the obvious way. (See Figure 4.5). Following the conventions of Section 3.4, we say that a generalized access path α is a *prefix* of a generalized access path β, denoted by $\alpha \leq \beta$, when there is a field path $\delta \in \Delta$, such that $\beta = \alpha.\delta$. We say that α is a *proper prefix* of β, denoted by $\alpha < \beta$, when $\delta \neq \epsilon$. The function $\cdot\cdot$ is lifted to handle sets of generalized access paths and sets of sequences of field identifiers.

In addition, we make use of the *flat* functional, well-known from functional programming. *flat* M returns the set of all elements of M, if M is a set of sets. Formally, $\mathit{flat}\ M \stackrel{\text{def}}{=} \{x \mid \exists A \in M : x \in A\}$.

4.4.1 Memory States

In this section, we define the representation of memory states in $\mathcal{LSL}$. Traditionally, a storeless semantics represents the heap by an equivalence relation over a set of access paths, where equivalence classes (implicitly) represent allocated objects. For readability, we use the equivalence classes directly, as we did in Section 3.4.

A *memory state* for a function p is a pair $\langle CPL_p, A_p \rangle$ of a set of cutpoint-labels, (denoted by CPL_p) and a heap (denoted by A_p). A heap is a finite (but unbounded) set of objects. An object (denoted by o) is described by a (possibly infinite) set of *generalized* access paths. Figure 4.3 gives the semantic domains used in $\mathcal{LSL}$ for a memory state of a function p.

A memory state $\langle CPL_p, A_p \rangle$ at a given point in an execution is composed of the labels of all the cutpoints of the current invocation (CPL_p) and a representation of the heap (A_p) at that point in the execution. To exclude states that cannot arise in any program, we now define the notion of *admissible states*.

r	$\in$	$Root_p = V_p \cup CPLbs_p$	Roots of generalized access paths
α, β	$\in$	$GAccPath_p = Root_p \times \Delta$	Generalized access paths
o	$\in$	$Obj_L^p = 2^{GAccPath_p}$	Objects
A, A_p	$\in$	$Heap_L^p = 2^{Obj_L^p}$	Heaps
$\sigma_L, \langle CPL_p, A_p\rangle$	$\in$	$\Sigma_L^p = 2^{CPLbs_p} \times Heap_L^p$	Memory state

Figure 4.3: Semantic domains of memory states for procedure p in $\mathcal{LSL}$. We use the syntactic domains V_p, $CPLbs_p$, and $GAccPath_p$ as semantic domains, too (and use italics font to denote a semantics value.)

Definition 4.4.1 (Admissible memory states) *A memory state $\langle \mathrm{CPL}_p, A_p\rangle$ for a function p at a given point in an execution is **admissible** iff*

(i) A generalized access path points-to (at most) one object, i.e., $\forall o, o' \in A_p$ if $o \neq o'$, then $o \cap o' = \emptyset$;
(ii) A_p is right-regular, i.e., $\forall o_1, o_2 \in A_p$ if $\alpha, \beta \in o_1$ and $\alpha.\delta \in o_2$ then $\beta.\delta \in o_2$;
(iii) A_p is prefix-closed, i.e., if $\alpha.f \in flat\ A_p$, then $\alpha \in flat\ A_p$; and
(iv) a root of every access path in the description of any object is either a local variable of p or a label of one of the cutpoints, i.e., if $\langle r, \delta\rangle \in flat\ A_p$ then either $r \in V_p$ or $r \in CPL_p$;
(v) $\emptyset \notin A$;
(vi) CPL_p satisfies the following requirements:
  (a) the cutpoint-labels in CPL_p are mutually disjoint,
  (b) CPL_p is right-regular (but not necessarily-prefix closed),
  (c) $\emptyset \notin CPL_p$.

The first three conditions are standard in storeless semantics, and were already used in Definition 3.4.1. The fourth condition limits the set of cutpoint-anchored paths that are tracked during an invocation to be rooted at a cutpoint of the invocation. The fifth condition is because we only represent objects that are pointed-to by a current or a pending access path. The sixth requirement captures the fact that the set of cutpoints is actually a subset of the objects in the heap when the function is invoked. Thus, CPL_p satisfies the first two requirements of heaps. However, because it is only a subset, it is not necessarily prefix-closed. The fact that the empty set is never in CPL_p is immediate once we recall that cutpoint-labels are generated only for objects that can be reached from the actual parameters when the function is invoked.

Because $\mathcal{LSL}$ preserves admissibility of states (see Lemma 4.5.6), in the sequel, whenever we refer to an $\mathcal{LSL}$ state, we mean an *admissible* $\mathcal{LSL}$ state.

It is possible to extract aliasing relationships from the sets of generalized access paths that describe the objects in a heap, and in particular to observe the heap structure as follows: a current variable `x` *points-to* an object o iff the access path $\langle x, \epsilon\rangle$ is in o. Similarly, cutpoint-label *cpl labels* object o iff $\langle cpl, \epsilon\rangle$ is in o. The field `f` of an object o_1 *points-to* object o_2 iff for every generalized access path $\langle r, \delta\rangle$ in o_1, the generalized access path $\langle r, \delta f\rangle$ is in o_2. A generalized access path α *points-to* (resp. *passes through*) an object o, if $\alpha \in o$ (resp. $\exists \beta < \alpha$ such that $\beta \in o$). An object o is *reachable* from a variable x, if there exists a field path $\delta \in \Delta$ such that $\langle x, \delta\rangle \in o$.

Example 4.4.2 The heap of the running example at the state in which `splice(t, z)` is invoked is shown in Figure 4.2($\sigma_L^{c\,4.2}$). It shows nine sets of generalized access paths. Each set represents one allocated list-element. At A^c, the caller's heap, access paths *x.n*, *t.n*, and *y* point-to the same object. The set of cutpoint-labels at the call site is empty. This is always the case for the main procedure. The second element in `x`'s list is a cutpoint for this invocation of `splice`: it is reachable from an actual parameter (its representation includes $t.n$) and by a local variable which is not aliased with an actual parameter (its representation includes y, but does not include either t or z).

The heap at the entry state to `splice`, shown in Figure 4.2($\sigma_L^{e\,4.2}$), differs from A^c in two ways: (i) the set of cutpoint-labels contains $\widehat{\{p.n\}}$, which labels the fourth element in the list; and (iii) objects are represented in terms of the generalized access paths that start either with p, q, or with $cpl = \widehat{\{p.n\}}$.

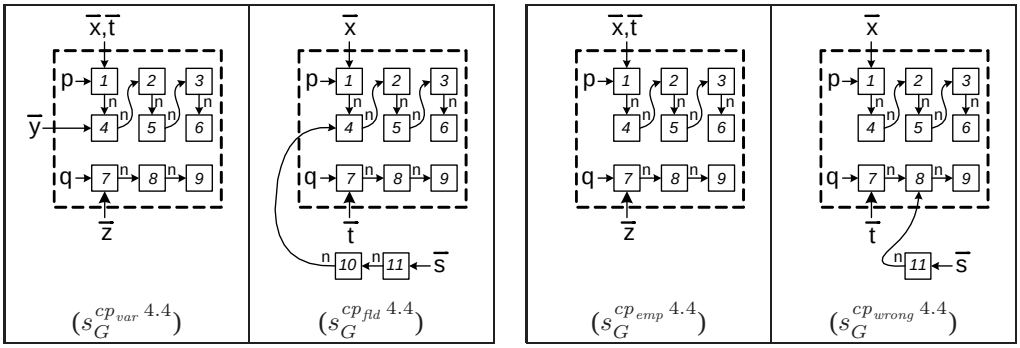


Figure 4.4: Potential *global-heaps* represented by the concrete local-heap $\sigma_L^{e\,4.2}$, shown at Figure 4.2. The local-heap of $\sigma_L^{e\,4.2}$ is depicted in the global-heap circumscribed with a dashed frame. Also, for clarity, we use the notation $\overline{\mathsf{x}}$ to denote a reference variable x of a pending call.

4.4.1.1 A Global View of Local-Heaps

The key reason for correctness of the semantics is that every local-heap represents all the *global* memory configurations containing that local-heap. Figure 4.4 illustrates that for the entry local-heap, shown at Figure 4.2($\sigma_L^{e\,4.2}$). We give two examples of potential global memory states represented by this local-heap and two examples which are not represented by this local-heap. Note that here, like in Section 3.3.2, we actually draw the global-heap, i.e., we draw all the allocated objects and we do *not* draw their storeless representation. We draw a dashed frame around the local-heap. Also, we use the notation $\overline{\mathsf{x}}$ to denote a reference variable x of a pending call.

The two left memory states are represented by the entry local-heap, shown at Figure 4.2($\sigma_L^{e\,4.2}$). Memory state $s_G^{cp_{var}\,4.4}$ is the one which actually occurs at the running example program. Memory state $s_G^{cp_{fld}\,4.4}$ is possible because the cutpoint is pointed-to by the n-field of the object pointed-to by s.n.n, an object which is irrelevant for the invocation. Notice that here the cutpoint is created via heap sharing and not stack sharing.

The two right stores represent impossible situations excluded by the cutpoint representation. In memory state $s_G^{cp_{emp}\,4.4}$, there are no cutpoints. In memory state $s_G^{cp_{wrong}\,4.4}$, there is one cutpoint but it is not the object pointed to by p.n at the procedure entry.

4.4.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{LSL}{\rightsquigarrow} \subseteq (\Sigma_L \times \mathit{stms}) \times \Sigma_L$. We give axioms for assignments and an inference rule for procedure calls in Figure 4.6 and Figure 4.7, respectively. All other statements are handled in the standard way. (See, e.g., [Kah87, NNH99]. Also, see Section D.1.)

To simplify notation, we assume A with a certain index (resp. prime) to be the heap component of a state σ_L with the same index (resp. prime). We use the same convention for indexed (or primed) versions of *CPL* and a state's cutpoint-labels component.

4.4.2.1 Helper Functions

We extend the functions $[\cdot]_\cdot$, $\mathit{rem}(\cdot,\cdot)$, and $\mathit{add}(\cdot,\cdot)$, defined in Figure 3.7 and explained in Section 3.4.2.1, to generalized access paths in the obvious way. (See Figure 4.5).

4.4.2.2 Atomic Statements

The *axioms* for atomic statements are given in Figure 4.6. We simplify the semantics by making the same assumptions as in Section 2.2.1.

The axioms for atomic statements in $\mathcal{LSL}$ are similar to the axioms in the $\mathcal{LSL}^{\text{CPF}}$ semantics (see Section 3.4.2.2), with the exception that they operate on $\mathcal{LSL}$'s memory states and not on $\mathcal{LSL}^{\text{CPF}}$'s memory states.

$.: GAccPath \times \Delta \to GAccPath$ s.t.
$\langle r, \delta \rangle . \delta' \stackrel{\text{def}}{=} \langle r, \delta\delta' \rangle$

$.: 2^{GAccPath} \times \Delta \to 2^{GAccPath}$ s.t.
$a.\delta \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a\}$

$.: 2^{GAccPath} \times 2^{\Delta} \to 2^{GAccPath}$ s.t.
$a.D \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a, \delta \in D\}$

$[]: GAccPath \times Heap_L \to Obj_L$ s.t.
$[\alpha]_A \stackrel{\text{def}}{=} \{\beta \in a \mid a \in A, \alpha \in a\}$

$rem: Heap_L \times 2^{GAccPath} \to Heap_L$ s.t.
$rem(A, a) \stackrel{\text{def}}{=} (map(\lambda o.o \setminus a.\{\delta \in \Delta\})\ A) \setminus \{\emptyset\}$

$add: Heap_L \times 2^{GAccPath} \times GAccPath \to Heap_L$ s.t.
$add(A, a, \alpha) \stackrel{\text{def}}{=} map(\lambda o.\, o \cup a.\{\delta \in \Delta \mid \alpha.\delta \in o\})\ A$

Figure 4.5: Helper functions of the operational semantics of $\mathcal{LSL}$.

$\langle \mathtt{x} = \mathtt{alloc}\ t, \langle \mathit{CPL}, A \rangle \rangle \overset{LSL}{\rightsquigarrow} \langle \mathit{CPL}, A \cup \{\{x\}\} \rangle$

$\langle \mathtt{x} = \mathtt{y}, \langle \mathit{CPL}, A \rangle \rangle \overset{LSL}{\rightsquigarrow} \langle \mathit{CPL}, add(A, \{x\}, y) \rangle$

$\langle \mathtt{x} = \mathtt{null}, \langle \mathit{CPL}, A \rangle \rangle \overset{LSL}{\rightsquigarrow} \langle \mathit{CPL}, rem(A, \{x\}) \rangle$

$\langle \mathtt{x} = \mathtt{y.f}, \langle \mathit{CPL}, A \rangle \rangle \overset{LSL}{\rightsquigarrow} \langle \mathit{CPL}, add(A, \{x\}, y.f) \rangle \qquad y \in \mathit{flat}\ A$

$\langle \mathtt{x.f} = \mathtt{null}, \langle \mathit{CPL}, A \rangle \rangle \overset{LSL}{\rightsquigarrow} \langle \mathit{CPL}, rem(A, [x]_A.f) \rangle \qquad x \in \mathit{flat}\ A$

$\langle \mathtt{x.f} = \mathtt{y}, \langle \mathit{CPL}, A \rangle \rangle \overset{LSL}{\rightsquigarrow} \langle \mathit{CPL}, \overline{add(A, [x]_A.f, y)} \rangle \qquad x \in \mathit{flat}\ A$

Figure 4.6: Axioms for atomic statements in the local-heap semantics. Note that the set of cutpoint-labels is not changed. The side-condition $x \in \mathit{flat}\ A$ (resp. $y \in \mathit{flat}\ A$) means that x's (resp. y) value is not *null*.

Note that the set of cutpoint-labels does not affect nor is affected by these axioms. Thus, we omit the description of the axioms. (A detailed description of the axioms is given in Section 3.4.2.2).

4.4.2.3 Procedure Calls

The *inference rule* for procedure calls is defined in Figure 4.7. The rule defines the program state σ_L^r that results from an invocation `y=p(x1,...,xk)` at memory state σ_L^c, assuming that the execution of the body of p at memory state σ_L^e results in memory state σ_L^x. The heaps A^c and A^r are described by sets of generalized access paths starting at the caller's variables and cutpoint-labels, whereas the heaps A^e and A^x are described by sets of generalized access paths that start at the callee's formal parameters, cutpoint-labels, and return variable. The rule provides the means to reconcile the different representations.

The rule uses the functions $Call_q^{y=p(x_1,\ldots,x_k)}$ and $Ret_q^{y=p(x_1,\ldots,x_k)}$, which are parameterized for each call statement in the program. $Call_q^{y=p(x_1,\ldots,x_k)}$ computes the memory state σ_L^e that results at the entry of p when $y = p(x_1,\ldots,x_k)$ is invoked by q in memory state σ_L^c. The caller's memory state after the invocation is restored by the function $Ret_q^{y=p(x_1,\ldots,x_k)}$. This function computes the memory state of the caller at the return-site (σ_L^r) according to q's memory state at the call-site (σ_L^c) and p's memory state at the exit-site (σ_L^x). In the rest of this section we describe the rule for an arbitrary call statement $y = p(x_1,\ldots,x_k)$ by an arbitrary procedure q. The rule utilizes additional helper functions, defined in Figure 4.8, which we gradually explain.

The main idea behind the rule is to utilize the fact that a procedure cannot modify objects that are not in its local-heap (i.e., in the part of the heap that is *not* reachable from any actual parameter when the procedure is invoked). In particular, because $\mathcal{LSL}$ describes objects in terms of the (generalized) access paths that point-to them, these "inaccessible" objects have the same description before and after the call. Thus, only the description of the objects in the procedure's local-heap (i.e., in the part of the heap that the procedure can access) is (possibly) updated. The update is carried out using the *cutpoints of the invocation*.[4] In essence, the semantics freezes the initial descriptions of the cutpoints and arranges for them to persist throughout the execution of the called procedure. This sets up a relation between values on entry to values on exit. At the return, the frozen information is used to update the description of objects in the called procedure's local-heap via an operation that is (roughly) similar to a relational join [Cod70]. (The operation is not a "pure" relational join because of some name adjustments that are needed due to the different representation of objects by the caller and by the callee.)

To find which objects are in the local-heap of the called procedure, i.e., reachable from the actual parameters ($x_1,\ldots,x_k$), we first compute the set of objects that are *pointed-to* by p's actual parameters (O_c^{args}). Then, the auxiliary function *RObjs* finds the part of the caller's heap (A^c) that is reachable from these objects (O_c^{passed}).

The description of the objects after the call should account for the mutations (destructive updates) of the heap performed by the callee. However, because the invoked procedure cannot modify objects that it cannot access, it can only modify fields of objects in O_c^{passed}. Thus, to compute the (possibly) updated description of objects in O_c^{passed} (as well as of objects that the callee allocates) it is sufficient to have a description of every object in O_c^{passed} (and of every object allocated by the callee) comprised of the (generalized) access paths that start at objects that separate O_c^{passed} from the rest of the caller's heap: When the procedure returns, we just replace any (generalized) access paths $\langle r_p, \delta_p\rangle$ in the description of every object in the heap of the callee (A^x) that start at a "separating object" o', by access paths of the caller $\langle r_q, \delta_q\delta_p\rangle$ such that $\langle r_q, \delta_q\rangle$ points-to o', but does not pass through O_c^{passed} (and thus cannot be modified). Technically, this is done as described below.

The auxiliary function $CPObjs_q$ (cf. Figure 4.8) determines the cutpoints for this procedure invocation (O_c^{cp}). Cutpoints are the objects that "separate" O_c^{passed} from the rest of the caller's heap. Recall that we do not consider objects that are pointed-to by actual parameters as cutpoints.[5] Thus, the function $CPObjs_q$, which is passed the caller's memory state as well as the previously computed O_c^{args} and O_c^{passed}, considers only objects in $O_{deep} = O_c^{passed} \setminus O_c^{args}$ as possible cutpoints. Following the intuition of cutpoints as "separating objects", an object $o \in O_{deep}$ is qualified as a cutpoint if (and only if) one of the following holds:

- o is pointed-to by a local variable of the caller (O_{vars}), or
- o is pointed-to by an object in the part of the caller's heap that is not passed to the function (O_{fld}), or

[4] The same mechanism is used to compute the description of objects that the callee allocates.

[5] The reason that we do not want to consider objects that are pointed-to by actual parameters as cutpoint is because we want to distinguish between the handling of updates through parameter objects, which is the simpler case which was already discussed in Chapter 3, to the handling of updates through the other objects which separates the callee's heap from that of the caller's.

$$\frac{\langle \text{body of } p, \sigma_L^e\rangle \overset{LSL}{\rightsquigarrow} \sigma_L^x}{\langle y = p(x_1,\ldots,x_k), \sigma_L^c\rangle \overset{LSL}{\rightsquigarrow} \sigma_L^r}$$

where

$\sigma_L^e = \mathit{Call}_q^{y=p(x_1,\ldots,x_k)}(\sigma_L^c)$
$\sigma_L^r = \mathit{Ret}_q^{y=p(x_1,\ldots,x_k)}(\sigma_L^c, \sigma_L^x)$

$O_c^{args} = \{[x_i]_{A^c} \mid 1 \le i \le k,\ [x_i]_{A^c} \neq \emptyset\}$
$O_c^{passed} = \mathit{RObjs}(A^c)\ O_c^{args}$
$O_c^{cp} = \mathit{CPObjs}_q(\langle \mathit{CPL}^c, A^c\rangle)\ (O_c^{args}, O_c^{passed})$
$bind_{args} = \lambda o \in O_c^{args}.\{\langle h_i, \epsilon\rangle \mid 1 \le i \le k,\ x_i \in o\}$
$bind_{cp} = \lambda o \in O_c^{cp}.\{\langle sub(bind_{args})\ o, \epsilon\rangle\}$
$bind_{call} = \lambda o \in O_c^{args} \cup O_c^{cp}. \begin{cases} bind_{args}(o) & o \in O_c^{args} \\ bind_{cp}(o) & o \in O_c^{cp} \end{cases}$

$Call_q^{y=p(x_1,\ldots,x_k)} : \Sigma_L^q \to \Sigma_L^p\ s.t.$
$Call_q^{y=p(x_1,\ldots,x_k)}(\langle \mathit{CPL}^c, A^c\rangle) \overset{\text{def}}{=} \langle map(sub(bind_{args}))\ O_c^{cp},\ map(sub(bind_{call}))\ O_c^{passed}\rangle$

$Ret_q^{y=p(x_1,\ldots,x_k)} : \Sigma_L^q \times \Sigma_L^p \to \Sigma_L^q\ s.t.$
$Ret_q^{y=p(x_1,\ldots,x_k)}(\langle \mathit{CPL}^c, A^c\rangle, \langle \mathit{CPL}^x, A^x\rangle) \overset{\text{def}}{=} \langle CPL^c, (A^c \setminus O_c^{passed}) \cup map(sub(bind_{ret}))\ A^x\rangle$

where

$bind_{ret} = \lambda a \in range(bind_{call}) \cup \{\{\langle ret, \epsilon\rangle\}\}.$
$\begin{cases} \{\langle y, \epsilon\rangle\} & a = \{\langle ret, \epsilon\rangle\} \\ Bypass(O_c^{passed}) \circ bind_{call}^{-1}(a) & \text{otherwise} \end{cases}$

Figure 4.7: The inference rule for procedure calls in $\mathcal{LSL}$. The rule is given for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q. We assume that the formal parameters of p are $h_1, \ldots, h_k$.

$$\textit{ROBjs}\colon \mathit{Heap}_L \to (2^{\mathit{Obj}_L} \to 2^{\mathit{Obj}_L}) \text{ s.t.}$$
$$\textit{ROBjs}(A)\ O \stackrel{\text{def}}{=} \{o \in A \mid o' \in O,\ \delta \in \Delta,\ o'.\delta \subseteq o\}$$

$$\mathit{Bypass}\colon 2^{\mathit{Obj}_L} \to (\mathit{Obj}_L \to 2^{\mathit{GAccPath}}) \text{ s.t.}$$
$$\mathit{Bypass}(O)\ o \stackrel{\text{def}}{=} \{\langle r, \delta\rangle \in o \mid \forall \delta' < \delta.\ \langle r, \delta'\rangle \notin \mathit{flat}\, O\}$$

$$\mathit{sub}\colon (2^{\mathit{GAccPath}} \to 2^{\mathit{GAccPath}}) \to (\mathit{Obj}_L \to 2^{\mathit{GAccPath}}) \text{ s.t.}$$
$$\mathit{sub}(\mathit{bind})\ o \stackrel{\text{def}}{=} \mathit{flat}\left\{ \mathit{bind}(a).\delta \;\middle|\; \begin{array}{c} a \in \mathit{dom}(\mathit{bind}), \\ \delta \in \Delta,\ a.\delta \subseteq o \end{array} \right\}$$

$$\mathit{CPObjs}_q\colon \Sigma^q_L \to (2^{\mathit{Obj}^q_L} \times 2^{\mathit{Obj}^q_L} \to 2^{\mathit{Obj}^q_L}) \text{ s.t.}$$
$$\mathit{CPObjs}_q(\langle \textit{CPL}^c, A^c\rangle)\ (O^{\mathit{args}}_c, O^{\mathit{passed}}_c) \stackrel{\text{def}}{=}$$
$$\begin{array}{l} \mathit{Let} \\ \quad O_{\mathit{deep}} = O^{\mathit{passed}}_c \setminus O^{\mathit{args}}_c \\ \quad O_{\mathit{vars}} = \{[\langle x, \epsilon\rangle]_{A^c} \in O_{\mathit{deep}} \mid x \in V_q\} \\ \quad O_{\mathit{fld}} = \left\{ o \in O_{\mathit{deep}} \;\middle|\; \begin{array}{c} \exists o' \in A^c \setminus O^{\mathit{passed}}_c, \\ \exists f \in \mathcal{F},\ o'.f \subseteq o \end{array} \right\} \\ \quad O_{\mathit{cpl}} = \{[\langle \mathit{cpl}, \epsilon\rangle]_{A^c} \in O_{\mathit{deep}} \mid \mathit{cpl} \in \mathit{CPL}^c\} \\ \mathit{in} \\ \quad O_{\mathit{vars}} \cup O_{\mathit{cpl}} \cup O_{\mathit{fld}} \end{array}$$

Figure 4.8: Helper functions for the procedure-call rule. The function CPObjs_q is parameterized for every procedure q in the program. Recall that V_q is the set of q's local variables. The functions *ROBjs*, Bypass, and sub are the extension to generalized access paths in the obvious way of the functions with the same names defined in Figure 3.10.

- o separates the heap of the *caller* from the heap of one of the pending calls, i.e., o is a cutpoint of the invocation of the caller (O_{cpl}).

Back in Figure 4.7 we define several binding mappings to bridge the gap between the two different representations of objects (in terms of access paths of the caller and in terms of access paths of the callee). The function $\mathit{bind}_{\mathit{args}}$ maps objects pointed-to by actual parameters to the set of "trivial" access paths that are made up of the corresponding formal parameters. The function $\mathit{bind}_{\mathit{cp}}$ maps every cutpoint (in the caller representation) to the set of access paths that start with a formal parameter of the caller and point-to that cutpoint at the entry to the procedure, i.e., $\mathit{bind}_{\mathit{cp}}$ maps a cutpoint to its label (see Section 4.3). To compute the label of a cutpoint o, we apply $\mathit{sub}(\mathit{bind}_{\mathit{args}})$. The latter denotes a function that replaces every access path that starts with an actual parameter $\langle x_i, \delta\rangle$ in the representation of o by an access path $\langle h_i, \delta\rangle$ that starts with the corresponding formal parameter. (sub is defined in Figure 4.8.) The $\mathit{bind}_{\mathit{call}}$ combines the previous two mappings trivially as they have disjoint domains.

Having defined these mapping functions, computing the memory state of p in which its body will be evaluated (i.e., the description of the heap at the procedure entry) is straightforward. The set of cutpoint-labels (CPL^e) is computed by applying $\mathit{bind}_{\mathit{cp}}$ to every cutpoint. The heap component (A^e) is constructed by applying $\mathit{bind}_{\mathit{call}}$ to every object in O^{passed}_c. Note that in the resulting description, objects are described by the set of (generalized) access paths that point-to them and start either at a formal parameter or at a cutpoint object.

To handle the return of procedure p, we use an additional binding, $\mathit{bind}_{\mathit{ret}}$. This mapping is the inverse of $\mathit{bind}_{\mathit{call}}$ (hence getting back to the caller's representation of the object) composed with the function $\mathit{Bypass}(O^{\mathit{passed}}_c)$, which filters out generalized access paths (of the caller) that *pass through* the part of the heap that p had access to (O^{passed}_c). In addition, it also takes care of replacing access paths starting with special variable ret with the same access paths starting with result variable y. Note that applying $\mathit{bind}_{\mathit{ret}}$ is well defined because CPL^x and CPL^e are equal (the callee cannot modify the set of objects that separate its own local-heap from the local-heap of of some pending call[6]).

[6]Note that in any transition $\langle \sigma_L, st\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$, the cutpoint-labels in σ_L and σ'_L are the same.

The cutpoint-labels component of the state after the return of p is the same as before the invocation (CPL^c) because the callee (p) cannot modify the set of objects that separate the heap of its caller (q) from the heap of some other (earlier) pending-call. The new heap is called A^r. It is derived by removing from the heap at the call-site the passed objects (O_c^{passed}), plugging in the heap that results from evaluating p's body (A^x), and substituting the description of all the objects by applying $sub(bind_{ret})$ to every object in A^x.

Example 4.4.3 Applying the procedure call rule for the invocation `t = splice(x, y);` in our running example results in the following sets and mappings:

$$\begin{array}{lcl} O_c^{args} & = & \{\{x\},\{y\}\} \\ O_c^{passed} & = & \{\{x\},\{x.n\},\{x.n.n\},\{y\},\{y.n\},\{y.n.n\}\} \\ O_c^{cp} = & = & \emptyset \\ bind_{args} & = & [\{x\} \mapsto \{p\}, \{y\} \mapsto \{q\}] \\ bind_{cp} & = & [] \\ bind_{ret} & = & \{\{x\} \mapsto \{p\}, \{y\} \mapsto \{q\}, \{ret\} \mapsto \{t\}\} \end{array}$$

Note that this invocation is cutpoint-free, thus O_c^{cp} is empty and $bind_{cp}$ is undefined. The computed values of the other sets and mappings is identical to their computed values in Example 3.4.6.

Example 4.4.4 Applying the procedure-call rule for the invocation `s = splice(t, z);` in our running example results in the following sets and mappings:[7]

$$\begin{array}{lcl} O_c^{args} & = & \{\{x,t\},\{z\}\} \\ O_c^{passed} & = & \left\{ \begin{array}{ccc} \{x,t\} & \{y.n, x.n^2, t.n^2\}, & \{y.n^3, x.n^4, t.n^4\}, \\ \{y, x.n, t.n\}, & \{y.n^2, x.n^3, t.n^3\}, & \{y.n^4, x.n^5, t.n^5\}, \\ \{z\}, & \{z.n\}, & \{z.n.n\} \end{array} \right\} \\ O_c^{cp} & = & \{\{y, x.n, t.n\}\} \\ bind_{args} & = & [\{y, x.n, t.n\} \mapsto \{p\}, \{z\} \mapsto \{q\}] \\ bind_{cp} & = & [\{x.n, t.n, y\} \mapsto \{\widehat{\{p.n\}}\}] \\ bind_{ret} & = & \{\{x,t\} \mapsto \{p\}, \{z\} \mapsto \{q\}, \{w\} \mapsto \{s\}, \{\widehat{\{p.n\}}\} \mapsto \{x.n, t.n, y\}\} \end{array}$$

Note that $\langle y, \epsilon \rangle \in O_{deep} = O_{passed} \setminus O_{args}$, thus $\{y, x.n, t.n\} \in \mathit{Cutpoints}$. This cutpoint is labeled by $\widehat{\{p.n\}}$, which provides a heap-context independent label for that cutpoint.

4.5 Properties of $\mathcal{LSL}$

In this section, we investigate the properties of the $\mathcal{LSL}$ semantics. In particular, we show that $\mathcal{LSL}$ is observationally equivalent to the standard semantics. Thus, abstractions of $\mathcal{LSL}^{\text{CPF}}$ can be used to conservatively verify properties of programs with respect to the standard semantics.

4.5.1 Observational Equivalence

The only means by which a program can observe a state is by access paths. In particular, the program cannot refer to the cutpoint-labels component of the state. To state the theorems, we need some preliminary definitions about access-path equality and observational equivalence. We use the same simplifying notational conventions as in Section 2.2.1. Note that in both semantics an access path is equal to `null` when it has a prefix which is equal to `null`.

Definition 4.5.1 (Access path equality) *Access paths α and β are **equal** in a given state σ_L, denoted by $[\![\alpha = \beta]\!]_{LSL}(\sigma_L)$, if $\forall a \in A.\, \alpha \in a \iff \beta \in a$. An access path α is **equal to null** in state σ_L, denoted by $[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L)$, if $\alpha \notin$ flat A.*

[7] The notation $x.n^k$ is explained in the caption of Figure 3.5.

Definition 4.5.2 (Observational equivalence) *Let p be a procedure. The states $\sigma_L \in \Sigma_L^p$ and $s_G \in \mathcal{S}_G^p$ are **observationally equivalent** if for all $\alpha, \beta, \gamma \in AccPath_p$,*

(i) $[\![\alpha = \beta]\!]_L(\sigma_L) \Leftrightarrow [\![\alpha = \beta]\!]_G(s_G)$*, and*

(ii) $[\![\gamma = \texttt{null}]\!]_L(\sigma_L) \Leftrightarrow [\![\gamma = \texttt{null}]\!]_G(s_G)$.

We also define observational equivalence between states in $\mathcal{LSL}$ in the same way.

The following theorem states that $\mathcal{LSL}$ is equivalent to $\mathcal{GSB}$, in the sense that both behave equivalently w.r.t. termination, and that execution of statements preserves observational equivalence.

Theorem 4.5.3 (Equivalence) *Let p be a procedure. Let $\sigma_L \in \Sigma_L^p$ and $s_G \in \mathcal{S}_G^p$ be observationally equivalent states. Let st be an arbitrary statement in p. The following holds:*

$$\langle st, \sigma_L \rangle \overset{LSL}{\rightsquigarrow} \sigma'_L \iff \langle st, s_G \rangle \overset{GSB}{\rightsquigarrow} s'_G.$$

Furthermore, σ'_L and s'_G are observationally equivalent.

We prove Theorem 4.5.3 by establishing a stronger property of the $\mathcal{LSL}$ semantics: the preservation of *Context-Aware Equivalence*. Informally, the *Context-Aware Equivalence* theorem shows that the cutpoints are, in a sense, the "store-based part" of $\mathcal{LSL}$: they are used to label and fix certain objects, something that is done automatically if we have locations. The theorem is formally stated and proved in Section D.2.2.

The following theorem states that $\mathcal{LSL}$ can be used to: (i) verify data-structure invariants that are expressed by access-path equalities at a program point; and (ii) assert the absence of *null*-valued pointer dereferences. Formally, a property is an invariant at a (labeled) statement if is satisfied in any memory-state that occurs just before the (labeled) statement is executed.

Corollary 4.5.4 *Let P be a program, p a procedure ,* lb *a program point in p. For any $\alpha, \beta \in AccPath_p$, $[\![\alpha = \beta]\!]_L$ is an invariant of P at* lb *iff $[\![\alpha = \beta]\!]_G$ is an invariant of P at* lb.

The following theorem states that $\mathcal{LSL}$ can detect memory leaks[8] without investigating reachability from *roots* of pending access paths. A memory leak can occur only when a variable or a field is assigned `null`. The "leaked objects" are the ones that are not pointed-to only by suffixes of the nullified variable (or field).

Corollary 4.5.5 *A memory leak can occur only when a variable or a field is assigned* `null`*. Furthermore,*

- *Executing a statement* `x = null` *in a memory state $\langle \text{CPL}, A \rangle$ leaks an object o iff $o \subseteq x.\Delta$.*
- *Executing a statement* `x.f = null` *in a memory state $\langle \text{CPL}, A \rangle$ leaks an object o iff $o \subseteq [\langle x, \epsilon \rangle]_A.f.\Delta$.*

4.5.2 Standard Properties

The following theorems state that the $\mathcal{LSL}$ semantics has certain standard properties.

The following lemma ensures that the $\mathcal{LSL}$ semantics preserves admissible states.

Lemma 4.5.6 (Admissibility) *Let st be a statement and σ_L an admissible state. If $\langle st, \sigma_L \rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$ then σ'_L is also an admissible state.*

Furthermore, $\mathcal{LSL}$ is a deterministic semantics; this holds because memory allocation is deterministic. (In contrast, most store-based semantics do not have a deterministic memory-allocation mechanism.)

[8] By a memory leak we mean an object that is not pointed-to by any access path; i.e., neither by an access path of the current call nor by one of a pending call.

Lemma 4.5.7 (Determinism) *Let st be a statement and σ_L an admissible state. If $\langle st, \sigma_L\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$ and $\langle st, \sigma_L\rangle \overset{LSL}{\rightsquigarrow} \sigma''_L$, then $\sigma'_L = \sigma''_L$.*

The following lemma states that $\mathcal{LSL}$ is fully abstract. To state this property, we use the notation $P[\cdot]$ for *program contexts*. The *denotation* $[\![st]\!]_L$ of a statement is defined to be the (partial) function $\lambda\sigma_L.\sigma'_L$ where $\langle \sigma_L, st\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$. This lemma holds because $\mathcal{LSL}$ has a canonic representation for memory states. Furthermore, the memory states are defined by terms of an equivalence relation between access paths. Thus, they represent only reachable parts of the memory state. As a result, if two memory states differ, then there exist certain access paths, including `null`, which are equal in one state, but not equal in the other.

Lemma 4.5.8 (Full Abstraction) *Let st_1 and st_2 be two statements such that for all program contexts $P[\cdot]$ and all states σ_L the states $[\![P[st_1]]\!]_L(\sigma_L)$ and $[\![P[st_2]]\!]_L(\sigma_L)$ are observationally equivalent. Then $[\![st_1]\!]_L = [\![st_2]\!]_L$.*

4.5.3 Heap Modularity

The following theorems state that $\mathcal{LSL}$ manipulates the heap in a "modular" way. Thanks to these properties, the *analysis* can also be heap-modular.

The following theorem states that a procedure has no effect on the observable properties of the unreachable part of the heap.

Theorem 4.5.9 (Framed Execution) *Let q be a procedure. Let $\sigma^c_L, \sigma^r_L \in \Sigma^q_L$ be states of procedure q such that $\langle \sigma^c_L, y = p(x_1, \ldots, x_k)\rangle \overset{LSL}{\rightsquigarrow} \sigma^r_L$. Let O^{passed}_c be the objects in σ^c_L that are reachable from $x_1, \ldots, x_k$. Let $\alpha, \beta, \gamma \in GAccPath_q \setminus y.\Delta$ be arbitrary generalized access paths of procedure q that do not start with y and do not **pass through** objects in O^{passed}_c. The following properties hold:*

(i) $[\![\alpha = \beta]\!]_{LSL}(\sigma^c_L) \iff [\![\alpha = \beta]\!]_{LSL}(\sigma^r_L)$, *and*

(ii) $[\![\gamma = \texttt{null}]\!]_{LSL}(\sigma^c_L) \iff [\![\gamma = \texttt{null}]\!]_{LSL}(\sigma^r_L)$.

Note that the above theorem is also applicable for access paths that *point-to* objects in the part of the heap that the procedure can access, but do not pass through this part.[9]

The following theorem states that a procedure cannot observe its context, i.e., that the execution of the procedure body is not affected by the cutpoint-labels component of the state.

Theorem 4.5.10 (Context Indifference) *Let p be a procedure. Let $\sigma^1_L, \sigma^2_L \in \Sigma^p_L$ be observationally equivalent states of p. Let st be an arbitrary statement in p. The following holds:*

$$\langle \sigma^1_L, st\rangle \overset{LSL}{\rightsquigarrow} \sigma^{1'}_L \iff \langle \sigma^2_L, st\rangle \overset{LSL}{\rightsquigarrow} \sigma^{2'}_L.$$

Furthermore, $\sigma^{1'}_L$ and $\sigma^{2'}_L$ are observationally equivalent.

The following theorem states that a procedure has a similar effect on contexts that differ only by the *contents* of the part of the heap that is not reachable from actual parameters. Practically speaking, this theorem justifies the reuse of the results of an analysis of a procedure invocation in "similar" contexts.

Theorem 4.5.11 (Heap Modularity) *Let p be a procedure. For $i = 1, 2$, let q_i be a procedure, $\sigma^{c_i}_L \in \Sigma^{q_i}_L$ be a state of procedure q_i, $y^i = p(x^i_1, \ldots, x^i_k)$ be a statement in procedure q_i, and $\sigma^{e_i}_L = Call^{y^i = p(x^i_1, \ldots, x^i_k)}_{q_i}(\sigma^{c_i}_L) \in \Sigma^p_L$ be the state that results at the entry to procedure p when it is invoked at $\sigma^{c_i}_L$. If $\sigma^{e_1}_L$ and $\sigma^{e_2}_L$ are observationally equivalent then the following properties hold:*

(i) $\langle \sigma^{c_1}_L, y^1 = p(x^1_1, \ldots, x^1_k)\rangle \overset{LSL}{\rightsquigarrow} \sigma^{r_1}_L \iff \langle \sigma^{c_2}_L, y^2 = p(x^2_1, \ldots, x^2_k)\rangle \overset{LSL}{\rightsquigarrow} \sigma^{r_2}_L$, *and*

[9] Recall that an access path α *passes through* an object o if there exists a proper prefix $\alpha' < \alpha$ such that α' points-to o.

(ii) if $CPL^{e_2} \subseteq CPL^{e_1}$ *and* $\langle \sigma_L^{e_1}, \textit{body of } p \rangle \overset{LSL}{\rightsquigarrow} \sigma_L^{x_1}$, *then*
$\sigma_{\mathbf{L}}^{\mathbf{r_2}} = \mathbf{gc}(Ret_{q_2}^{y^2=p(x_1^2,\ldots,x_k^2)}(\sigma_L^{c_2}, \sigma_{\mathbf{L}}^{\mathbf{x_1}}))$, *where* $\mathbf{gc}(\langle \text{CPL}, A \rangle) \overset{\text{def}}{=} \langle \text{CPL}, A \backslash \emptyset \rangle$.

We need to apply **gc** to the heap produced by $Ret_{q_2}^{y^2=p(x_1^2,\ldots,x_k^2)}(\sigma_L^{c_2}, \sigma_{\mathbf{L}}^{\mathbf{x_1}})$ because of the following technical reason: it is possible that some of the objects in A^{x_1} are reachable only from objects that are cutpoints when p is invoked at $\sigma_L^{c_1}$ but not when it is invoked at $\sigma_L^{c_2}$. Thus, some objects that are reachable (i.e., pointed-to by a current or a pending access path) at $\sigma_L^{r_1}$ might not be reachable at $\sigma_L^{r_2}$.

4.6 Interprocedural Shape Analysis: An Overview

This section provides an informal overview of our approach for interprocedural functional shape analysis. The presentation is given at a semi-technical level; a more detailed treatment of this material, as well as several elaborations on the ideas covered here, is presented in the later parts of this chapter: Section 4.7 describes a shape abstraction of $\mathcal{LSL}$; Section 4.8 describes the abstract transformers; and Section 4.9 presents our implementation and experimental evaluation.

Our algorithm tabulates abstract memory configurations before and after procedure calls. Our algorithm is an extension of the framework of [SRW02]. Technically, abstract memory configurations are represented by *3-valued* logical structures and the abstract transformers are derived from their specification in the *concrete* semantics. (See Section 2.5).

4.6.1 Handling Cutpoint-Free Procedure Calls

In this section, we describe the way cutpoint-free invocations are handled by the interprocedural shape analysis. We note that for cutpoint-free invocations, the algorithm presented in this chapter is quite similar to the algorithm presented in Chapter 3.

4.6.1.1 Concrete States

Concrete memory configurations (concrete states) are represented by 2-valued logical structures. They are drawn as directed graphs, following the graphical notations introduced in Section 2.5.1.3.

> **Example 4.6.1** Figure 4.1(c) shows the concrete memory states that occur at the invocation `t = splice(x, y);` from `main`. (At this point, please ignore $\widehat{p}$, $\widehat{q}$, $\widehat{n}$, and the circle nodes. Their role is explained later on in this section.)
>
> Our concrete semantics uses procedure local-heaps, thus it does not pass the linked list pointed-to by `z` to the procedure because the latter is not reachable from either one of the actual parameters: `x` or `y`. As we have already seen, the use of local-heaps allows the analyzer to infer the effect of a procedure in a context independent way, and thus to potentially increase the scalability of the analysis.

4.6.1.2 Abstract States

Abstract states are 3-valued logical structures. They are drawn as directed graphs, following the graphical notations introduced in Section 2.5.1.3.

Nodes in abstract states are shown annotated by properties of represented allocated objects. Moreover, concrete objects with different properties are represented by different nodes. The property r_x holds for objects that are reachable from a variable `x` via a sequence of pointer fields. Thus, e.g., at the call state to `t = splice(x, y);`, depicted in Figure 4.1($\sigma_L^{c^{4.1}}$), the second and the third concrete elements of the list pointed-to by `x` are represented by the summary node annotated by r_x in the abstracted state. We can see that the abstract memory state abstract the actual data values and the lengths of the lists.

Example 4.6.2 Figure 4.1(d) shows the abstract memory states which conservatively represent the concrete memory states which may occur at the invocation `t = splice(x, y);`, depicted in Figure 4.1(c). (Again, for now, please ignore $\widehat{p}$, $\widehat{q}$, $\widehat{n}$, and the circle nodes):

- Figure 4.1($S^{c\,4.1\sharp}$) depicts the abstract call memory state. This abstract memory state represents all memory state containing three disjoint lists of length 2 or more with heads, `x`, `y`, and `z`, respectively.
- Figure 4.1($S^{e\,4.1\sharp}$) depicts the relevant part of the heap at the entry to the callee: two disjoint lists of length 2 or more pointed to by `p` and by `q`, respectively.
- Figure 4.1($S^{x\,4.1\sharp}$) depicts the store at the exit—`q` points to the second element of the list pointed to by `p`. Note how the tail of the list is reachable from both `p` and `q`.
- Figure 4.1($S^{r\,4.1\sharp}$) depicts the memory state upon return.

4.6.1.3 Discussion

Because the number of properties such as r_x is finite for a given program, so is the number of nodes and shape-graphs. Therefore, by simple tabulation of input/output shape-graphs, we obtain a context- and flow-sensitive analysis which enforces matching calls and returns even in the presence of recursion. Moreover, the fact that on a procedure call, we only represent the local-heap reachable from actual parameters allows us to abstract facts at the caller which are irrelevant to the callee. For example, the memory states shown in Figure 4.1($S^{e\,4.1\sharp}$) resp. Figure 4.1($S^{x\,4.1\sharp}$) also represents entry resp. exit memory states that occur in recursive calls. (Note that the recursive calls are invoked when `pn` points to the second element of the list pointed to by `p`. This allows the analyzer to increase reuse of procedure summaries because it does not discriminate between contexts with different irrelevant parts of the heap. In fact, the cost of handling recursive `splice` is propositional to the cost of handling the iterative version and in both cases we can prove that the result is indeed an unshared acyclic list (see Section 4.9.1).

4.6.2 Handling Procedure Calls with Cutpoints

We are now ready to explain our treatment of cutpoints, and the role of $\widehat{p}$, $\widehat{q}$, and the circle nodes. Figure 4.2(c) shows the concrete memory states that occur at the invocation `s = splice(t, z);` from `main`. Figure 4.2(d) shows the corresponding abstract memory states.

This call differs from the first call because `y` points-to the second element in the list which was passed to the procedure. Therefore, destructive updates in the procedure may make nodes (un)reachable from `y` and thus change the r_y property. In other words, `y` creates stack sharing into the local-heap. The challenge is finding a way to update properties such as reachable-from-`y` without explicitly representing `y`. The ability to do so is crucial for reusing the analysis of the procedure body across different procedure invocations.

Our solution uniformly treats stack sharing and sharing of fields from the rest of the heap by recording *sharing patterns* into the local-heap. The main idea is to give special treatment to *cutpoints* (see Definition 3.3.2). Our analysis follows $\mathcal{LSL}$, and labels cutpoints with access paths that points to them and start with formal parameters at procedure-entry. This provides a naming scheme for the cutpoint-labels (recording the sharing pattern) which is independent of the irrelevant context. For example, we label the list-element pointed-to by `y` with `p.n`.

We use circle nodes, which we refer to as *object-labels*, to represent the values of access paths emanating from formal parameters at procedure entry. Every object o in the local-heap passed to the callee has a corresponding "shadow" circle node which freezes the values of o's pointer-fields. The value of pointer variable `p` at procedure-entry is denoted by $\widehat{p}$. The value of pointer field `f` at procedure-entry is denoted by $\widehat{f}$. The "shadow" of a cutpoint is connected to the cutpoint object with an *lbl*-labeled edge. For readability, we depict cutpoint objects with an incoming double-line arrow. Note that our labeling scheme records the sharing pattern in a way which ignores the *contents* of irrelevant context. This ensures that the analysis results are applicable for every calling context which generates the same sharing pattern.

4.6.2.1 Concrete States

Consider the concrete state at the entry state of the invocation `s = splice(t, y);`, depicted in Figure 4.2(c). The second node in `p`'s list is a cutpoint. Its label is the second circle node on $\widehat{p}$'s list (the asterisk marks the fact

that this label corresponds to a cutpoint). The lbl-edge connects the label to the cutpoint node. This signifies that the access path p.n points-to a cutpoint object. It is used to relate the access-path's value between the call and the return.

Now the code at the procedure body is executed without the need to explicitly represent y or any other irrelevant part of the global-heap but instead with labels marking the cutpoint objects.

Cutpoint-labels are frozen between procedure entry and procedure exit. Moreover, without loss of generality we assume that formal parameters refer to the same objects before and after the call. Therefore, the semantics can reconnect y to the second node from $\widehat{p}$ when the procedure exits. Thus, the mutation of the heap is transmitted to the caller while correctly updating y.

4.6.2.2 Abstract States

In the interprocedural shape analysis we tabulate shape-graphs with special summary labels potentially representing multiple cutpoints. In the example, there are no (circle) summary object-labels nodes. Specifically, because the procedure only includes one cutpoint object, the latter is not summarized. Therefore, upon exit, the abstract transformer can precisely update reachability from y.

In those cases where cutpoint objects are summarized, our analysis is sound but may be overly conservative. Moreover, this may also degrade performance.

4.7 A Shape Abstraction of $\mathcal{LSL}$

In this section, we define a shape abstraction of $\mathcal{LSL}$ using *canonical abstraction* [SRW02]. The new abstraction forms the basis for a new interprocedural shape analysis algorithm for programs with cutpoints, described in Sections 4.8 and 4.9.

An interesting aspect of our abstraction is that it does not abstract single memory states, but *pairs of memory states*. Specifically, it abstracts every memory state that occurs during the execution of a procedure p together with the memory state that occurs at the entry to the procedure. Technically, the entry memory state is "freezed" and used to encode the cutpoint-labels and it is not modified throughout the execution of the procedure. (We note that our choice to record the cutpoint-labels by the "freezed" entry state comes from a pragmatic reason explained in Remark 4.7.4).

Technically, 3-valued logical structures are used to represent (pairs of) unbounded memory states. The tracked properties are encoded as predicates.

We define a Galois connection between the powerset domain of pairs of $\mathcal{LSL}$ memory states and $3Struct$ using a *representation function* (see Section 2.5.1) $\beta_L \colon \Sigma_L \times \Sigma_L \to 3Struct$, which maps a *pair* of program states to a *single 3-valued* logical structure which conservatively represents *both* states.

Function β_L is a composition of two functions:

(i) *to2VLS*: $\Sigma_L \times \Sigma_L \to 2Struct$, which maps a pair of local-heap (possibly with cutpoints) $\sigma_L^e \in \Sigma_L$ and $\sigma_L \in \Sigma_L$ to an unbounded *2-valued* logical structure S. (The memory state $\sigma_L^e = \langle \mathit{CPL}^e, A^e \rangle$ is expected to be the memory state at the entry to the procedure in which $\sigma_L = \langle \mathit{CPL}, A \rangle$ occurs. Thus, it is expected that $\mathit{CPL} = \mathit{CPL}^e \subseteq simple(A^e)$, where $simple(A)$ removes the cutpoint-anchored access paths from the representation of every object in A.)

(ii) *canonical abstraction*: $2Struct \to 3Struct$, which conservatively bounds S.

The Galois connection

$$(2^{\Sigma_L \times \Sigma_L}, \alpha \colon 2^{\Sigma_L \times \Sigma_L} \to 2^{3Struct}, \gamma \colon 2^{3Struct} \to 2^{\Sigma_L \times \Sigma_L}, 2^{3Struct})$$

is defined as:

$$\alpha(\mathit{CC}) = \left\{ \beta_L(\sigma_L^e, \sigma_L) \in 3Struct \;\middle|\; \begin{array}{c} \langle \sigma_L^e, \sigma_L \rangle \in \mathit{CC}, \\ \sigma_L^e = \langle \mathit{CPL}^e, A^e \rangle,\ \sigma_L = \langle \mathit{CPL}^e, A \rangle, \\ CP^e \subseteq simple(A^e) \end{array} \right\}$$

$$\gamma(\mathit{AA}) = \left\{ \langle \sigma_L^e, \sigma_L \rangle \in \Sigma_L \times \Sigma_L \;\middle|\; \begin{array}{c} \beta_L(\sigma_L^e, \sigma_L) \sqsubseteq S^\sharp \in \mathit{AA}, \\ \sigma_L^e = \langle \mathit{CPL}^e, A^e \rangle,\ \sigma_L = \langle \mathit{CPL}^e, A \rangle, \\ CP^e \subseteq simple(A^e) \end{array} \right\}$$

$\mathit{to2VLS}^p : \Sigma^p_L \times \Sigma^p_L \to \mathit{2Struct}_p$ s.t.
$\mathit{to2VLS}^p_{\langle \mathit{CPL}^e, A^e\rangle}(\langle \mathit{CPL}, A\rangle) = \langle U, \iota\rangle$
where
$U = A \uplus \mathit{simple}(A^e)$
where $\mathit{simple}(A^e) = \mathit{map}(\lambda o.o \cap (F_p.\Delta))\ A^e$

$$
\begin{array}{lcl}
\iota(x)(v) & = & v \in A \text{ and } x \in v \\
\iota(f)(v_1, v_2) & = & v_1 \in A, v_2 \in A \text{ and } v_1.f \subseteq v_2 \\
\iota(eq)(v_1, v_2) & = & v_1 = v_2 \\
 & & \\
\iota(r_x)(v) & = & v \in A \text{ and } \exists \alpha \in v \text{ s.t. } \langle x, \epsilon\rangle \leq \alpha \\
\iota(ils)(v) & = & v \in A \text{ and } \exists \alpha.n \in v,\ \beta.n \in v \text{ s.t. } [\alpha]_A \neq [\beta]_A \\
\iota(c)(v) & = & v \in A \text{ and } \exists \alpha \in v,\ \beta \in v \text{ s.t. } \alpha < \beta \\
 & & \\
\iota(\mathit{isObj})(v) & = & v \in A \\
\iota(\mathit{isLabel})(v) & = & v \in \mathit{simple}(A^e) \\
\iota(\widehat{x})(v) & = & v \in \mathit{simple}(A^e) \text{ and } x \in v \\
\iota(\widehat{f})(v_1, v_2) & = & v_1 \in \mathit{simple}(A^e),\ v_2 \in \mathit{simple}(A^e) \text{ and } v_1.f \subseteq v_2 \\
\iota(\mathit{lbl})(v_1, v_2) & = & v_1 \in \mathit{simple}(A^e), v_1 \in \mathit{CPL}^e, \text{ and } \langle v_1, \epsilon\rangle \in v_2 \\
 & & \\
\iota(\mathit{cp})(v) & = & \exists r \in \mathit{simple}(A^e) \text{ s.t. } r \in \mathit{CPL}^e \text{ and } \langle r, \epsilon\rangle \in v \\
\iota(r_{\mathit{cp}})(v) & = & \exists r \in \mathit{simple}(A^e),\ \delta \in \Delta \text{ s.t. } r \in \mathit{CPL}^e \text{ and } \langle r, \delta\rangle \in v
\end{array}
$$

Figure 4.9: The function $\mathit{to2VLS}^p$ maps (pairs of) memory states of procedure p in Σ^p_L to 2-valued logical structures. The memory state $\sigma^e_L = \langle \mathit{CPL}^e, A^e\rangle$ is expected to be the memory state at the entry to the procedure in which $\sigma_L = \langle \mathit{CPL}, A\rangle$ occurs. Thus, it is expected that $\mathit{CPL} = \mathit{CPL}^e \subseteq \mathit{simple}(A^e)$, where $\mathit{simple}(A)$ removes the cutpoint-anchored access paths from the representation of every object in A.

where $\beta_L(\sigma^e_L, \sigma_L) \sqsubseteq S^\sharp$ means that $S^\sharp \in \mathit{3Struct}$ conservatively represents $\beta_L(\sigma^{,}_L \sigma_L) \in \mathit{2Struct}$. (See Definition 2.5.4).

4.7.1 Representing Pairs of $\mathcal{LSL}$ Memory States by 2-Valued Logical Structures

The function *to2VLS*, defined in Figure 4.9, maps a pair of memory states $\langle \sigma^e_L, \sigma_L\rangle = \langle\langle U^e, \iota^e\rangle, \langle U, \iota\rangle\rangle$ of a procedure p to a *2-valued* logical structure S.

Heap allocated objects are represented by individuals. We note that the main reason for recording heap allocated objects from the entry state (σ^e_L) is as a way to represent the (sets of access paths comprising the) cutpoint-labels of the *current* program state (σ_L).

Tracked properties of the memory state are recorded by predicates. We allow every procedure p to be associated with a set of predicates $\mathcal{P}_p \subseteq \mathcal{P}$. We track properties for every procedure using this set of predicates, i.e., a logical structure representing a memory state of a procedure p defines (only) the meaning of the predicates in the set $\mathcal{P}_p \subseteq \mathcal{P}$. (See Section 4.7.1.2).

In this chapter, we use the predicates given in Figure 4.10 and Figure 4.11, which we gradually explain. In addition, we use the predicates inUc and inUx, new, and $\mathit{instance}$, shown in Figure 4.12 to implement the call rule. The role of these predicates is explained in Section 4.8.2.2.[10]

4.7.1.1 Representing Cutpoint-Labels

We represent pairs of memory states $\langle \sigma^e_L, \sigma_L\rangle = \langle\langle \mathit{CPL}^e, A^e\rangle, \langle \mathit{CPL}, A\rangle\rangle \in \Sigma_L \times \Sigma_L$ using a first-order 2-valued logical structure $S = \langle U, \iota\rangle \in \mathit{2Struct}$. The main challenge here is the representation of the *access paths*

[10] Recall that the link structure of the entry state defines the cutpoint-labels of the invocation. Also, see Remark 4.7.4.

Predicate	Intended Meaning
$x(v)$	reference variable x points to object v
$f(v_1, v_2)$	f-field of object v_1 points to object v_2
$eq(v_1, v_2)$	v_1 and v_2 are the same object or the same label
$isObj(v)$	v is a heap-allocated object
$isLb_O(v)$	v is an object-label
$isLb_{CP}(v)$	the object-label v is a *cutpoint*-label
$\widehat{x}(v)$	v labels the object that is pointed-to by the formal parameter x when the *current* procedure is invoked
$\widehat{f}(v_1, v_2)$	v_2 labels the object that is the f-successor of the object labeled by v_1 when the *current* procedure is invoked
$lbl(v_1, v_2)$	object v_2 is labeled by cutpoint-label v_1

Figure 4.10: Predicates used to represent memory states.

Predicate	Intended Meaning	Defining Formula
$r_{obj}(v_1, v_2)$	v_2 is reachable from object v_2 by following some field path	$isObj(v_1) \wedge isObj(v_2) \wedge F^*(v_1, v_2)$
$ils(v)$	v is *locally shared*. i.e., v is pointed-to by a field of more than one object in the *local-heap*	$isObj(v) \wedge \exists v_1, v_2 \colon v_1 \neq v_2 \wedge$ $isObj(v_1) \wedge isObj(v_2) \wedge F(v_1, v) \wedge F(v_2, v)$
$c(v)$	v resides on a directed cycle of fields	$\exists v_1 \colon F(v, v_1) \wedge F^*(v_1, v)$
$r_x(v)$	v is reachable from variable x	$isObj(v) \wedge \exists v_x \colon isObj(v_x) \wedge x(v_x) \wedge F^*(v_x, v)$
$cp(v)$	v is a cutpoint	$\exists v_l \colon isLb_{CP}(v_l) \wedge lbl(v_l, v)$
$r_{cp}(v)$	v is reachable by following some field-path from a cutpoint	$\exists v_l : isLb_{CP}(v_l) \wedge isObj(v) \wedge$ $\exists v_{cp} \colon isObj(v_{cp}) \wedge lbl(v_l, v_{cp}) \wedge F^*(v_{cp}, v)$

Figure 4.11: The instrumentation predicates. Predicates r_{obj}, ils, c, and r_x have similar meaning to the corresponding predicates in Figure 2.11(a). Their defining formulae utilizes the predicate $isObj$ as a way to emphasize that it may hold only for objects, and not for object-labels. (Technically, in addition, predicate $isObj$ is used to ensure that predicate r_{obj} holds for (u, u) iff u represents a heap-allocated object).

Predicate	Intended Meaning
$new(v)$	v is a newly created individual
$instance(v_1, v_2)$	v_1 is an instance of v_2
$inUc(v)$	v is a member of the caller's call-site universe
$inUx(v)$	v is a member of the callee's exit universe

Figure 4.12: Auxiliary predicates for universe-altering operations.

comprising every cutpoint-label. We address this issue by "freezing" the entry memory state (σ_L^e) in the current memory state of the procedure (σ_L).[10]

4.7.1.2 Tracked Properties

We represent memory states using the predicates shown in Figure 4.10 and Figure 4.11. The predicates listed above the double line in Figure 4.10 resp. Figure 4.11 are a rather straightforward adaptation of the predicates listed in Figure 2.11(a) resp. Figure 2.11(b), and were described in Section 2.5.1.2.[11]

We explain the role of the other predicates in the following paragraphs.

Core Predicates. Figure 4.10 lists the core predicates that we use.

- The predicates $isObj(v)$, $isLb_{CP}(v)$, and $isLb_O(v)$ distinguish individuals that represent heap-allocated objects, cutpoint-labels, and access paths, respectively. (Note that predicate $isObj$ holds for an individual u iff the predicate $isLb_O$ does not hold for u. We chose to use both predicates for clarity. Also note that if the predicate $isLb_{CP}$ holds for u then the predicate $isLb_O$ also holds for u). In the following, we refer to individuals used to represent access paths as *object-labels*.
- A predicate $\widehat{x}(v)$ records the object that was pointed-to by a formal parameter `x` when the current procedure was invoked. Similarly, a predicate $\widehat{f}(v_1, v_2)$ records the value of an `f`-field when the current procedure was invoked.
- The binary predicate $lbl(v_1, v_2)$ relates a node v_1 that represents a cutpoint-label to the node v_2 that represents the corresponding cutpoint.

Instrumentation Predicates. Instrumentation predicates record derived properties of individuals, and are defined using a logical formula over core predicates. They are used to refine the abstract semantics (see Section 2.5.1). Figure 4.11 lists the instrumentation predicates used in this section.

- We use $\widehat{F}(v_1, v_2)$ as a shorthand notation for $\bigvee_{f \in FieldId_P^\star} \widehat{f}(v_1, v_2)$ in addition to the shorthand notations $F(v_1, v_2)$ and $\varphi^*(v_1, v_2)$, which were introduced in Figure 3.15.[12]
- The unary predicate cp records the property that a list element is a cutpoint. The unary predicate r_{cp} records the property that a list element is reachable by a cutpoint-anchored path.
- The predicates cp and r_{cp} are used to record information regarding cutpoint-anchored paths in a similar manner to the way h and r_h record information regarding access-paths. However, unlike local variables, the number of cutpoints is unbounded. Thus, we cannot have a predicate recording the reachable list-elements from every cutpoint. Instead, we use individuals to represent cutpoint-labels, and "mark" cutpoint objects with the cp predicate.

> **Example 4.7.1** *2-valued* logical structures are depicted as directed graphs. We use the following graphical notations, in addition to the ones described in Example 2.5.2. A directed edge between nodes u_1 and u_2 that is labeled with binary predicate symbol p indicates that $\iota^S(p)(u_1, u_2) = 1$. We draw a node u that represents an object (i.e., $\iota^S(isObj)(u) = 1$) as a box and a node that represents a label (i.e., $\iota^S(isLabel)(u) = 1$) as a circle. We draw cutpoint objects with an incoming double-line arrow. Cutpoint-labels are marked with an asterisk. We depict the value of a pointer variable `x` by drawing an edge from `x` to the node that represents the object that `x` points-to. For a pointer parameter `q`, we also draw an edge from $\widehat{q}$ to the *object-label* of the object that `q` points-to.
>
> Figure 4.1(c) shows the 2-valued logical structures pertaining to the $\mathcal{LSL}$ memory states at the call-site, entry-site, exit-site, and return-site during the invocation `t = splice(x, y)` in the running example, shown in Figure 4.1(b). Specifically, $S^{c\,4.1} = \mathit{to2VLS}_{\sigma_L^\emptyset}(\sigma_L^{c\,4.1})$, $S^{e\,4.1} = \mathit{to2VLS}_{\sigma_L^{e\,4.1}}(\sigma_L^{e\,4.1})$, $S^{x\,4.1} = \mathit{to2VLS}_{\sigma_L^{e\,4.1}}(\sigma_L^{x\,4.1})$, and $S^{r\,4.1} = \mathit{to2VLS}_{\sigma_L^\emptyset}(\sigma_L^{r\,4.1})$.

[11] In addition to the instrumentation predicates listed in Figure 2.11(b), we also use the binary instrumentation predicate $r_{obj}(v_1, v_2)$ which records reachability between objects.

[12] We remind that $F(v_1, v_2)$ is a shorthand for $\bigvee_{f \in FieldId_P^\star} f(v_1, v_2)$ and that for a formula φ with two free variables, the notation $\varphi^*(v_1, v_2)$ is a shorthand for the reflexive transitive closure of φ, i.e., $\varphi^*(v_1, v_2) \stackrel{\text{def}}{=} eq(v_1, v_2) \vee (TC\ w_1, w_2 : \varphi(w_1, w_2))(v_1, v_2)$.

The call state and the return state are memory states of the `main` procedure. They are abstracted together with the (empty) memory state $\sigma_L^{\emptyset} = \langle \emptyset, \emptyset \rangle$ at the entry to `main`.

The invocation `t = splice(x, y);` is cutpoint-free. Thus, the set of cutpoint-labels is empty in every memory state that arises during the invocation. (Consider, for example, the entry memory state and the exit memory state, depicted in Figure 4.1($\sigma_L^{e\,4.1}$) and Figure 4.1($\sigma_L^{x\,4.1}$), respectively). As a result, in any 2-valued memory state that represents a pair of memory state that arises during cutpoint-free invocation, the individuals representing objects from the different memory states are separated. Specifically, there are no lbl-edges.

Example 4.7.2 Figure 4.2(c) shows the 2-valued logical structures pertaining to the memory states at the call-site, entry-site, exit-site, and return-site during the invocation `s = splice(t, z)` in the running example, shown in Figure 4.2(b). Specifically, $S^{c\,4.2} = \mathit{to2VLS}_{\sigma_L^{\emptyset}}(\sigma_L^{c\,4.2})$, $S^{e\,4.2} = \mathit{to2VLS}_{\sigma_L^{e\,4.2}}(\sigma_L^{e\,4.2})$, $S^{x\,4.2} = \mathit{to2VLS}_{\sigma_L^{e\,4.2}}(\sigma_L^{x\,4.2})$, and $S^{r\,4.2} = \mathit{to2VLS}_{\sigma_L^{\emptyset}}(\sigma_L^{r\,4.2})$.

The invocation `s = splice(t, y);` is not cutpoint-free. Specifically, the second element in `t`'s list is a cutpoint. Thus, the set of cutpoint-labels in, e.g., the entry memory state, depicted in Figure 4.2($\sigma_L^{e\,4.2}$) contains the cutpoint-label $cpl = \{\widehat{p.n}\}$.

Cutpoint-labels are represented in *2-valued* logical structures by individuals. These individuals are depicted as circle node labeled with an asterisk ($*$).

The (only) access path $\widehat{p.n}$ leading to the cutpoint is represented by a chain of nodes. Each node in the chain is depicted by a circle. The first circle in the chain represents the (object-label of the) list element pointed to by `p` at the entry state. It is depicted by an edge emanating from the $\widehat{p}$. It's $\widehat{n}$-successor point to the node representing the (cutpoint-label of the) cutpoint. The labeling of the cutpoint object is depicted by an lbl-labeled edge emanating from the cutpoint-label to the cutpoint object.

The unary predicate cp records the property that a list element is a cutpoint. The unary predicate r_{cp} records the property that a list element is reachable by a cutpoint-anchored path.

For example, r_{cp} holds for all the nodes in the tail of `p`'s list at the entry state and for the tail of `q`'s list at the exit state.

4.7.1.3 Admissible Memory States

Not all *2-valued* logical structures represent memory states that are compatible with the semantics of EAlgol (or JAVA or C, for that matter). For example, in EAlgol, each pointer variable points to at most one heap-allocated element. To exclude states that cannot arise in any program, we now adapt the notion of *admissible* 2*-valued logical structures*, introduced in Definition 3.6.3, to local-heaps with cutpoints.

Definition 4.7.3 (Admissible 2-Valued Logical Structures) *A* 2-valued *logical structure $S = \langle U, \iota \rangle$ representing a local-heap for a procedure p at a given point in an execution is* ***admissible*** *iff*

(i) *S is admissible according to Definition 3.6.3.*

(ii) *Every node $u \in U$ represents either an* object *or an* object-label, *i.e.,* $S \models \mathit{isObj}(u) \iff \neg \mathit{isLb}_O(u)$. *Furthermore,* cutpoint-labels *must be* object-labels, *i.e.,* $S \models \mathit{isLb}_{CP}(u) \implies \mathit{isLb}_O(u)$.

(iii) *A frozen variable points to at most one node, i.e., for all nodes $u, u_1, u_2 \in U$, $S \models \bigvee_{x \in F_p} \widehat{x}(u_1) \wedge \widehat{x}(u_2) \implies \mathit{eq}(u_1, u_2)$. Furthermore, variables only point to objects while* frozen *variables only point to* object-labels, *i.e.,* $S \models \bigvee_{x \in V_p} x(u) \implies \mathit{isObj}(u)$ *and* $S \models \bigvee_{x \in F_p} \widehat{x}(u) \implies \mathit{isLb}_O(u)$.

(iv) *A frozen field is a partial function, i.e., for every triple of nodes $u, u_1, u_2 \in U$, $S \models \bigvee_{f \in FieldId_P^\star} \widehat{f}(u, u_1) \wedge \widehat{f}(u, u_2) \implies \mathit{eq}(u_1, u_2)$. Furthermore, a field maps objects to objects and objet-labels to object-labels, i.e.,* $S \models \bigvee_{f \in FieldId_P^\star} f(u_1, u_2) \implies (\mathit{isObj}(u_1) \wedge \mathit{isObj}(u_2))$ *and* $S \models \bigvee_{f \in FieldId_P^\star} \widehat{f}(u_1, u_2) \implies (\mathit{isLb}_O(u_1) \wedge \mathit{isLb}_O(u_2))$.

(v) *lbl is a function that maps cutpoint-labels to objects, i.e., for every nodes $u, u_1, u_2 \in U$, $S \models \mathit{lbl}(u_1, u_2) \implies \mathit{isLb}_{CP}(u_1) \wedge \mathit{isObj}(u_2)$. Furthermore, lbl is an injective function, i.e., $S \models \mathit{lbl}(u, u_1) \wedge \mathit{lbl}(u, u_2) \implies \mathit{eq}(u_1, u_2)$ and $S \models \mathit{lbl}(u_1, u) \wedge \mathit{lbl}(u_2, u) \implies \mathit{eq}(u_1, u_2)$. In addition*

$lbl|_{\{u\in U:\iota(isLb_{CP})(u)=1\}}$ is a surjective function, i.e., if $S \models isLb_{CP}(u)$ then there exists a node $u_{cpo} \in U$ such that $S \models lbl(u_{cpo}, u)$.

(vi) Every object-label is reachable from at least one frozen variable, i.e., for every node $u_{lbl} \in U$, if $S \models isLb_O(u_{lbl})$ then there exists a node $\hat{u}$ such that $\hat{u}$ is an object-label, i.e., $S \models \hat{u}$, and $S \models \bigvee_{x\in X} \widehat{x}(\hat{u}) \wedge \widehat{F}^*(\hat{u}, u_{lbl})$.

Remark 4.7.4 *We note that the reason we record the cutpoint-labels by "freezing" the entry state is a pragmatic one: We wish not to lose information when we encode an $\mathcal{LSL}$ memory state using a 2-valued logical structure. In particular, we wish not to lose information regarding the contents of cutpoint-labels. The content of a cutpoint-label forms a regular language. Thus, we record it using a finite-state-machine-like construction comprised of object-labels. We chose to implement this construction by freezing the link structure of the entry state. This choice gives us a* simple and deterministic *way to define the mapping from (pairs of) $\mathcal{LSL}$ memory states to 2-valued logical structures. (We note, however, that alternative choices exist. For example, it is possible to record every cutpoint-label using the minimal deterministic finite state machine which accepts its contents.)*

4.7.2 Conservatively Representing Memory States of $\mathcal{LSL}$ by 3-Valued Logical Structures using Canonical Abstraction

We obtain a *bounded* conservative representation of (unbounded) *2-valued* logical structures using canonical abstraction. (See Section 2.5.1.3.)

Example 4.7.5 Figure 4.1(d) depicts the 3-valued logical structure that results by applying *canonical abstraction* to the 2-valued logical structures representing the memory states at the call-site, entry-site, exit-site, and return-site at the invocation `t = splice(x, y);` in the running example, shown in Figure 4.1(c). Specifically, $S^{c\sharp}_{4.1}$ is a canonical abstraction of $S^{c}_{4.1}$, $S^{e\sharp}_{4.1}$ is a canonical abstraction of $S^{e}_{4.1}$, $S^{x\sharp}_{4.1}$ is a canonical abstraction of $S^{x}_{4.1}$, and $S^{r\sharp}_{4.1}$ is a canonical abstraction of $S^{r}_{4.1}$.

(*3-valued* logical structures are depicted using the graphical conventions introduced in Example 2.5.5.)

The universe of $S^{e}_{4.2}$ contains 12 nodes. The only nodes that have the same values for all the unary predicates are the last two nodes in the tail of `p`'s list and the two nodes in the tail of the list pointed to by `q`. In addition, all the objects-labels that are not pointed to by frozen formal parameters have the same unary properties. Thus, the universe of $S^{e\sharp}_{4.2}$ contains 7 nodes: 4 nodes pointed to by parameters resp. frozen parameters, 2 summary nodes representing list elements and 1 summary node representing object-labels.

We see, in two ways, that any memory state represented by $S^{e\sharp}_{4.2}$ results from a cutpoint-free invocation: (i) the lbl does not relate any object-label with this list-element, and (ii) the unary predicate cp does not hold for any list element.

Example 4.7.6 Figure 4.2(d) depicts the 3-valued logical structure that results by applying *canonical abstraction* to the 2-valued logical structures representing the memory states at the call-site, entry-site, exit-site, and return-site at the invocation `s = splice(t, z);` in the running example, shown in Figure 4.2(c). Specifically, $S^{c\sharp}_{4.2}$ is a canonical abstraction of $S^{c}_{4.2}$, $S^{e\sharp}_{4.2}$ is a canonical abstraction of $S^{e}_{4.2}$, $S^{x\sharp}_{4.2}$ is a canonical abstraction of $S^{x}_{4.2}$, and $S^{r\sharp}_{4.2}$ is a canonical abstraction of $S^{r}_{4.2}$.

The universe of $S^{e}_{4.2}$ contains 18 nodes. The only nodes that have the same values for all the unary predicates are the last four nodes in the tail of `p`'s list and the two nodes in the tail of the list pointed to by `q`. In addition, besides the object-labels pointed to by frozen formal parameters and the cutpoint-label, all the other 6 objects-labels have the same unary properties. Thus, the universe of $S^{e\sharp}_{4.2}$ contains one more node than the universe of $S^{e\sharp}_{4.1}$, the additional node represents the cutpoint object.

We see that in any memory state represented by $S^{e\sharp}_{4.2}$ there is one cutpoint. The fact that the second element in `p`'s list is a cutpoint is recorded in two ways: (i) the lbl related the cutpoint-label (depicted

$$\langle st, XS\rangle \overset{LCP^\sharp}{\rightsquigarrow} \{\beta_L(\sigma^e_L, \sigma'_L) \mid \langle \sigma^e_L, \sigma_L\rangle \in \gamma(XS), \langle st, \sigma_L\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L\}$$

Figure 4.13: A specification of the abstract inference rules for atomic statements.

by the circle node labeled with an asterisk) with this list-element, and (ii) the unary predicate cp holds (only) for this list element. (A double-line arrow marks nodes for which the predicate cp holds).

4.8 Abstract Transformers

In this section, we define the abstract transformers used by the analysis. Following the presentation in Section 3.7, we first provide in Section 4.8.1 a declarative specification of the abstract transformers in a non-algorithmic fashion: We specify the abstract semantics using the *best abstract transformer* [CC79] (see Section 2.5.2.2). In Sections 4.8.2 and 4.8.3, we utilize the framework of [SRW02] to obtain conservative abstract transformers: We define the effect of intraprocedural statements as well as call and return statements using first order formulas with transitive closure and show how to compute the (abstract) effect of program statements.

4.8.1 A Declarative Specification of the Abstract Transformers

The meaning of statements is described by a transition relation $\overset{LCP^\sharp}{\rightsquigarrow} \subseteq (3Struct \times st) \times 3Struct$. In this section, we provide a declarative specification of the meaning of statements, given by "abstract" inference rules in the same style as the natural semantics. The abstract inference rules operate on *3-valued* logical structures. (See Section 2.5.2.2).

Figure 4.13 and Figure 4.14 show the specification of the abstract inference rules for atomic statements and procedure-calls, respectively. These rules are given in the declarative style of the best abstract transformer [CC79]: Every abstract inference rule emulates a corresponding concrete inference rule using represented states (see Figure 2.16).

Remark 4.8.1 *Note the first component of the pair of memory states is computed once a procedure is invoked and never changes until it terminates.*

Example 4.8.2 Figure 4.2(d) shows an application of the procedure-call inference rule from Figure 4.14 to the invocation `s = splice(t, z);` in our running example on the (singleton) set $XS_q = \{S^{c\,4.2\sharp}\}$:

(i) $S^{c\,4.2\sharp}$ conservatively represents all the memory states that may arise at the call site to the invocation `s = splice(t, y);`. (More precisely $S^{c\,4.2\sharp}$ represents all pairs of $\mathcal{LSL}$ memory states whose first component is the empty memory state and whose second component is a memory state in which `z` points to an acyclic list with 3 or more elements and `x` and `t` point to another acyclic list which has 4 or more elements and whose second element is pointed to by `y`).

(ii) $S^{e\,4.2\sharp}$ conservatively represents all pairs of memory states $\langle \sigma^f_L, \sigma^e_L\rangle$ where (i) σ^f_L is an entry state in which `q` points to a list with 1 or more elements and `p` points to a list with 2 or more elements whose second element is a cutpoint, and (ii) $\sigma^e_L = Call_q^{s=splice(t,y)}(\sigma^c_L)$ is a memory state that may arise at the entry to `splice` when invoked on a call state σ^c_L represented by $S^{c\,4.2\sharp}$.

(iii) $S^{x\,4.2\sharp}$ conservatively represents all pairs $\langle \sigma^f_L, \sigma^x_L\rangle$ such that (i) σ^f_L is as described above and (ii) σ^x_L represents an exit state that may arise when splice starts executing from an entry state σ^e_L such that $\langle \sigma_L, \sigma^e_L\rangle \in \gamma(S^{e\,4.2\sharp})$.

(iv) $S^{r\,4.2\sharp}$, the structure *computed* at the return-site, which represent all memory states that may arise after `s=splice(t, y)` is invoked on a memory state represented by $S^{c\,4.2\sharp}$.

$$\frac{\langle \textit{body of } p, XS_p\rangle \overset{LCP^\sharp}{\rightsquigarrow} XS'_p}{\langle y = p(x_1,\ldots,x_k), XS_q\rangle \overset{LCP^\sharp}{\rightsquigarrow} XS'_q}$$

where

$$XS_p = \left\{ \beta_L(\sigma^e_L, \sigma^e_L) \;\middle|\; \begin{array}{c} \langle \sigma^{e^q}_L, \sigma^c_L\rangle \in \gamma(XS_q) \\ \sigma^e_L = Call_q^{y=p(x_1,\ldots,x_k)}(\sigma^c_L) \end{array} \right\}$$

$$XS'_q = \left\{ \beta_L(\sigma^{e^q}_L, \sigma^r_L) \;\middle|\; \begin{array}{c} \langle \sigma^{e^q}_L, \sigma^c_L\rangle \in \gamma(XS_q) \\ \sigma^e_L = Call_q^{y=p(x_1,\ldots,x_k)}(\sigma^c_L) \\ \langle \sigma^e_L, \sigma^x_L\rangle \in \gamma(XS'_p) \\ \sigma^r_L = Ret_q^{y=p(x_1,\ldots,x_k)}(\sigma^c_L, \sigma^x_L) \end{array} \right\}$$

Figure 4.14: A specification of the abstract inference rules for procedure calls. The functions $Call_q^{y=p(x_1,\ldots,x_k)}$ and $Ret_q^{y=p(x_1,\ldots,x_k)}$ are defined in Figure 4.7. Note that we apply $Ret_q^{y=p(x_1,\ldots,x_k)}$ only for *compatible* pairs of memory states. Two pairs of memory states $\langle \sigma^{e^q}_L, \sigma^c_L\rangle$ and $\langle \sigma^e_L, \sigma^x_L\rangle$ are compatible when the sharing pattern that results from the invocation of p at σ^c_L (as recorded in σ^x_L) matches σ^e_L, the description of the context in σ^x_L, the state of p at the exit-site.

In $S^{x\,4.2\sharp}$, the lists pointed to by p and q are spliced together. As a result, at the exit-site the cutpoint object is now reachable from q and the tail of q's list is now reachable from the cutpoint. Specifically, the cutpoint becomes the n-successor of q. Therefore, even though y is not explicitly represented in $S^{x\,4.2\sharp}$, the inference rule allows us to conclude that at $S^{r\,4.2\sharp}$, the return-site's logical structure, the list element pointed to by y becomes the n-successor of q. Similarly, the list-element pointed to by y remains reachable from p, but it is no longer its n-successor. To conclude, definite values of many of the tracked properties of y can be established after the procedure call returns.

4.8.2 $\mathcal{LCP}$: A Concrete Localized-Heap Semantics based on 2-Valued Logic

In this section, we present $\mathcal{LCP}$, a concrete semantics that serves as the basis for our abstraction. The semantics records object-labels and cutpoint-labels as introduced in Section 4.6.2. Technically, we use first-order logical structures to represent local-heaps, and show how to realize the declarative definition of the abstract semantics, given in Section 4.8.1, as operations on first-order logical structures. For simplicity, we do not provide full formal details of these operations here. The formal details are given in Section D.3.1.

$\mathcal{LCP}$ represents pairs of memory states: It represents every memory state that may occur during the execution of a procedure p together with the memory state that occurs at the entry to the procedure when it was invoked. The entry memory state is used to encode the cutpoint-labels and it is not modified throughout the execution of the procedure.

The meaning of statements is described by a transition relation $\overset{LCPF^\sharp}{\rightsquigarrow} \subseteq (3Struct \times st) \times 3Struct$ that specifies how a statement st transforms an incoming logical structure into an outgoing logical structure.

4.8.2.1 Atomic Statements

The meaning of assignments is specified, primarily, by defining the values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [SRW02]. The inference rules for assignments are rather straightforward and can be found in Appendix D.3.1. For control statements, we use the standard rules of natural semantics, see, e.g., [Kah87, NNH99].

4.8.2.2 Procedure Calls

The procedure call rule for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q is given in Section D.3. The rule is instantiated for each call statement in the program. The rule is quite technically complicated, thus here we only describe its operation in a more informal way.

Our treatment of procedure calls and returns in $\mathcal{LCPF}$ follows their treatment in $\mathcal{LSL}$, and could be briefly described as follows:

(i) the call rule constructs the memory state at the callee's entry site (S_e) and
(ii) the caller's memory state at the call site (S_c) and the callee's memory state at the exit site (S_x) are used to construct the caller's memory state at the return site (S_r).

We now formally define and explain these steps.

Computing the Entry State

When a procedure p is called, the semantics *extracts* a relevant object-labeled local-heap from the heap provided at the call-site. To realize this operation using logical structures, our semantics takes the following steps: (i) creates new object-labels for all relevant objects (objects reachable from actual parameters of p). This is done by using the function **clone** defined in Figure 4.17 to clone all objects reachable from parameters of p. (ii) adjusts the values of label-predicates of the form $\widehat{x}(v)$ and $\widehat{f}(v_1, v_2)$. (iii) adjusts the values of cutpoint-labels predicate. (iv) removes all irrelevant objects by using the function **remove** defined in Figure 4.17.

Example 4.8.3 Figure 4.1($S^{e\,4.1}$) depicts the *2-valued* logical structure at the entry to the invocation `t = splice(x, y);` resulting during the application of the call rule to the call memory state, depicted in Figure 4.1($S^{c\,4.1}$). (See Example 4.7.1).

Example 4.8.4 Figure 4.2($S^{e\,4.2}$) depicts the *2-valued* logical structure at the entry to the invocation `s = splice(t, z);` resulting during the application of the call rule to the call memory state, depicted in Figure 4.2($S^{c\,4.2}$). (See Example 4.7.2).

Computing the Return State

Returning from a call to procedure p, the semantics *combines* the structure representing the caller memory-state at the call-site and the structure representing the memory-state of the callee at the exit-site. To realize this operation using logical structures, our semantics takes the following steps:

(i) it combines the structure representing the local-heap at the exit from p with the structure at the call site. This is done by using the function **combine** defined in Figure 4.17.
(i) uses cutpoints to merge the local-heap back into the global-heap by finding matching individuals. Technically, this is achieved by using extended transitive closure (transitive closure of pairs) to traverse matching paths in the caller's heap and the callee's local-heap. (See Section D.3.1.2).
(i) removes all labels of the local-heap, retaining only those labels originally present in the caller's heap.

Step (i) above is accomplished using the **combine** function. Figure 4.15 depicts the combined structures resulting during the application of the procedure call rule for the invocations `t = splice(x, y);` and `s = splice(t, y);` in our running example.

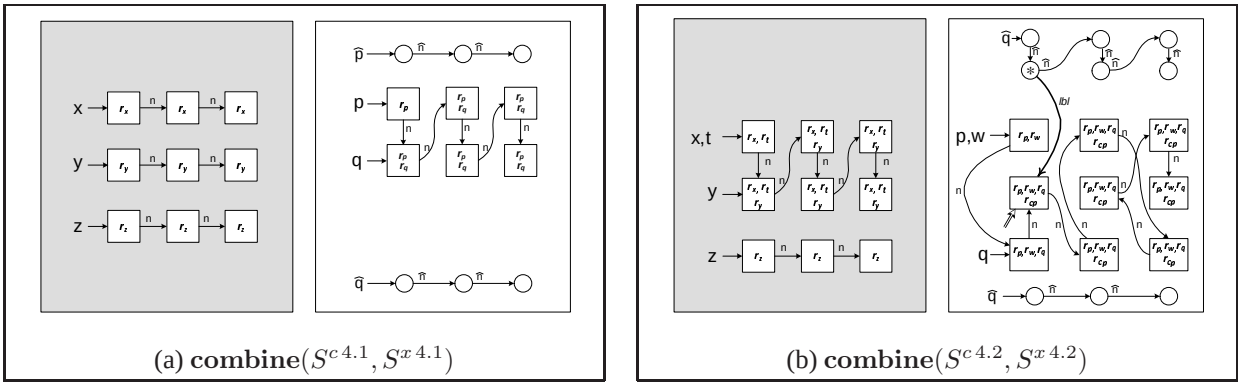
Step (ii) above is accomplished using the formulae shown in Figure 4.18. The formula $match$ is of special importance. It holds only for nodes v_1 and v_2 that meet the following conditions:

- v_1 and v_2 represent the same object o: v_1 represents o in the caller's local-heap at the memory-state in which the function was invoked; v_2 represents o in memory-state at the exit-site of the callee.
- o separates the caller's local-heap from the callee's local-heap, i.e., either o is pointed-to by a parameter ($match$) or it is a cutpoint of the invocation ($matchCP$).

Note that the formula $samePath$ (used by $matchPaths$) is the *only* extended-transitive-closure formula that we use.

The following example demonstrates how the formulae of Figure 4.18 are used to combine structures upon return of of the invocation `s = splice(t, y)` in the running example.

Figure 4.2(c)($S^{e\,4.2}$) depicts the *2-valued* logical structure at the entry to the invocation `s = splice(t, z);` resulting during the application of the call rule to the call memory state, depicted in Figure 4.2(c)($S^{c\,4.2}$). (See Example 4.7.2).



(a) **combine**$(S^{c\,4.1}, S^{x\,4.1})$ (b) **combine**$(S^{c\,4.2}, S^{x\,4.2})$

Figure 4.15: Combined structures. Individuals originating from the same state are circumscribed with a rectangular frame. Individuals originating from the call state are depicted against a shaded background. (a) the combined structures resulting from applying **combine** to the call state to `t = splice(x, y);`, $S^{c\,4.1}$, and the resulting exit state $S^{x\,4.1}$. Both structures are depicted in Figure 4.1. (b) the combined structures resulting from applying **combine** to the call state to `s = splice(t, y);`, $S^{c\,4.2}$, and the resulting exit state $S^{x\,4.2}$. Both structures are depicted in Figure 4.2.

Example 4.8.5 Consider the exit memory state to the invocation `s = splice(t, z);` shown in Figure 4.2($S^{x\,4.2}$). Upon return, this structure is combined with the structure at the caller's call-site, shown in Figure 4.2($S^{c\,4.2}$), resulting with the return memory state, shown in Figure 4.2($S^{r\,4.2}$).

Our operational semantics for procedure return updates the value of the predicate $y(v)$ (corresponding to a reference variables `y`) using the following update formulae:

$$\begin{aligned} y'(v) = isObj(v) \wedge ((inUc(v) \wedge y(v) \wedge \neg R_{\{t,z\}}(v)) \vee \\ (inUx(v) \wedge \exists v_1 \colon y(v) \wedge inUc(v_1) \wedge R_{\{t,z\}}(v_1) \wedge \\ match_{main,\{\langle p,t\rangle,\langle q,z\rangle\}}(v_1, v)) \end{aligned}$$
$$\text{where} \quad R_X(v) \stackrel{\text{def}}{=} \bigvee_{x \in X} \exists v_1 . x(v_1) \wedge F^*(v_1, v).$$

In the 2-valued logical which results from the combination of structures $S^{c\,4.2}$ and $S^{x\,4.2}$, this formula holds for the third individual of the list starting from `p`, as we are able to match the paths from the cutpoint-label and the paths from `y`.

The predicates $x(v)$, $z(v)$, and $t(v)$ are updated similarly, but fall to the simpler matching case ($match_{bind}(v_1, v_2)$).

Note that the operations used to model procedure call and return are operations that change the universe of a logical structure. We call such operations "universe altering". The universe altering functions use the helper functions **extend** and **project**, defined in Figure 4.17. The operation **extend** extends the partial interpretation mapping by providing a default value of 0 for objects in O for which ι is undefined. The operation **project** restricts the interpretation to be defined only for objects in a given set O.

The effects of the universe-altering functions are as follows:

- **clone**, duplicates all individuals that satisfy a given formula, and extends the interpretation accordingly. To record information about the newly allocated individuals, we use two additional auxiliary predicates new and $instance$ (defined in Figure 4.12). These predicates record newly created individuals and the source object for each cloned object, respectively.
- **remove**, removes all individuals that satisfy a given formula, and restricts the interpretation accordingly.
- **combine**, combines two given structures into a single structure and extends the interpretation accordingly. To distinguish individuals that come from different universes we use the auxiliary predicates $inUc$ and $inUx$, defined in Figure 4.12.

$$\mathbf{extend}(\iota, O)(p^k)(u_1, \ldots, u_k) \stackrel{\text{def}}{=}$$
$$\begin{cases} \iota(p)(u_1, \ldots, u_k) & : \ \langle u_1, \ldots, u_k \rangle \in dom(\iota(p)) \\ 0 & : \ u_i \in O \text{ for some } 0 < i \leq k \text{ and for all } 0 < j \leq k, u_j \in O \cup dom(\iota(p)) \\ \text{undefined} & : \ \text{otherwise} \end{cases}$$
$$\mathbf{project}(\iota, O)(p^k)(u_1, \ldots, u_k) \stackrel{\text{def}}{=}$$
$$\begin{cases} \iota(p)(u_1, \ldots, u_k) & : \ \{u_1, \ldots, u_k\} \subseteq O \\ \text{undefined} & : \ \text{otherwise} \end{cases}$$

Figure 4.16: Interpretation manipulating functions.

$\mathbf{clone}$: $\mathcal{WFF}_1 \times \mathit{2Struct} \to \mathit{2Struct}$ s.t.

$\mathbf{clone}(\varphi, \langle U, \iota \rangle) \stackrel{\text{def}}{=} \langle U', \iota' \rangle$ where

$O_{dup} = \{u.2 \mid u \in U, [\![\varphi(v)]\!]_2^{\langle U, \iota \rangle}(\langle v \mapsto u, \emptyset \rangle) = 1\}$

$U' = \{u.1 \mid u \in U\} \cup O_{dup}$

$\iota''(p)(u.1, \ldots, u_k.1) = \iota(p)(u_1, \ldots, u_k)$

$\iota' = \mathbf{extend}(\iota'', O_{dup})$

$$\left[\begin{array}{l} \iota'(new)(v) \stackrel{\text{def}}{=} v = u, \\ \iota'(instance)(w, v) \stackrel{\text{def}}{=} w = u.1 \text{ and } v = u.2 \end{array} \right]$$

$\mathbf{combine}$: $\mathit{2Struct} \times \mathit{2Struct} \to \mathit{2Struct}$ s.t.

$\mathbf{combine}(\langle U^1, \iota^1 \rangle, \langle U^2, \iota^2 \rangle) \stackrel{\text{def}}{=} \langle U^{1.1} \cup U^{2.2}, \iota' \rangle$ where

$U^{1.1} = \{u.1 \mid u \in U^1\}$

$\iota^{1.1}(p)(u_1.1, \ldots, u_k.1) = \iota^1(p)(u_1, \ldots, u_k)$

$U^{2.2} = \{u.2 \mid u \in U^2\}$

$\iota^{2.2}(p)(u_1.2, \ldots, u_k.2) = \iota^2(p)(u_1, \ldots, u_k)$

$\iota' = (\mathbf{extend}(\iota^{1.1}, U^{2.2}) \ \vee \ \mathbf{extend}(\iota^{2.2}, U^{1.1}))$

$$\left[\ inUc(u) \stackrel{\text{def}}{=} u = w.1, inUx(u) \stackrel{\text{def}}{=} u = w.2 \ \right]$$

$\mathbf{remove}$: $\mathcal{WFF}_1 \times \mathit{2Struct} \to \mathit{2Struct}$ s.t.

$\mathbf{remove}(\varphi, \langle U, \iota \rangle) \stackrel{\text{def}}{=} \langle U \setminus O, \iota' \rangle$ where

$O = \{u \in U \mid [\![\varphi(v)]\!]_2^{\langle U, \iota \rangle}(\langle v \mapsto u, \emptyset \rangle) = 1\}$

$\iota' = \mathbf{project}(\iota, U \setminus O)$

Figure 4.17: Universe altering functions. $\mathcal{WFF}_1$ denotes the set of well-formed formulae in first-order logic with transitive closure that have a single free variable v.

Shorthand	**Formula**
$match_{p,bind}(v_1, v_2)$	$inUc(v_1) \wedge isObj(v_1) \wedge inUx(v_2) \wedge isObj(v_2) \wedge$ $(match_{bind}(v_1, v_2) \vee matchCP_{p,bind}(v_1, v_2))$
$match_{bind}(v_1, v_2)$	$\bigvee_{\langle h,z\rangle \in bind} z(v_1) \wedge h(v_2)$
$matchCP_{p,bind}(v_1, v_2)$	$isCP_{p,\{z\mid\langle h,z\rangle\in bind\}}(v_1) \wedge \bigwedge_{\langle h,z\rangle\in bind} (R_{\{z\}}(v_1) \implies matchPaths_{h,z}(v_1, v_2))$
$matchPaths_{h,z}(v_1, v_2)$	$\forall v_z\colon inUc(v_z) \wedge isObj(v_z) \wedge$ $\forall l_h\colon inUx(l_h) \wedge isLb_O(l_h) \wedge \forall l_2\colon inUx(l_2) \wedge isLb_{CP}(l_2) \wedge$ $(z(v_z) \wedge \widehat{h}(l_h) \wedge lbl(l_2, v_2) \implies samePath(v_z, v_1, l_h, l_2))$
$samePath(v_x, v_{cp}, l_x, l_{cp})$	$(TC\ v_1, v_2; w_1, w_2 : \bigvee_{f\in FieldId^\star_P} f(v_1, v_2) \wedge \widehat{f}(w_1, w_2))\ (\ v_x, v_{cp}; l_x, l_{cp})$
$isCP_{p,X}(v)$	$R_X(v) \wedge \bigwedge_{x\in X} \neg x(v) \wedge$ $(\bigvee_{y\in V_p} y(v) \vee \exists v_1.\neg R_X(v_1) \wedge F(v_1, v) \vee \exists v_1.isLb_{CP}(v_1) \wedge lbl(v_1, v))$

Figure 4.18: Shorthand notations for formulae used to match individuals and paths when combining structures on procedure return. $FieldId^\star_P$ denotes the set of all the pointer-valued fields that are used in the program. V_p denotes the set of all local-variables (including formal parameters) of procedure p.

4.8.3 $\mathcal{LCP}^\sharp$: An Abstract Localized-Heap Semantics based on 3-Valued Logic

In this section, we present a conservative abstract semantics abstracting the $\mathcal{LCP}$ concrete semantics introduced in Section 4.8.2.

4.8.3.1 Abstract States

We conservatively represent multiple concrete memory states $SS \subset 2Struct$ by a single 3-valued logical structure $S^\sharp \in 3Struct$ using canonical abstraction [SRW02]. (See Definition 2.5.4).

We conservatively represent all object-labels resp. cutpoint-labels using a single abstract object-label resp. abstract cutpoint-label, with the exception of the object-labels pointed-to by (frozen) formal parameters, which are represented by a unique node.

We remind the reader that the Galois connection $(2^{\Sigma_L\times\Sigma_L}, \alpha\colon 2^{\Sigma_L\times\Sigma_L} \to 2^{3Struct}, \gamma\colon 2^{3Struct} \to 2^{\Sigma_L\times\Sigma_L}, 2^{3Struct})$ between pairs of memory states of $\mathcal{LSL}$ and 3-valued logical structures is obtained by composing β_L, which maps pairs of memory states of $\mathcal{LSL}$ to 3-valued logical structures, and canonical abstraction. See Section 4.7.

Remark 4.8.6 *We note that for the rather crude cutpoint-abstraction that we suggest in this section, the information used for the* bounded *abstraction of cutpoint-labels is readily available in the abstracted state (the second component of the abstracted pair). Thus, it is possible to define this abstraction using a Galois connection between the powerdomain of $\mathcal{LSL}$ memory states and the powerdomain of 3-valued logical structures instead of using a Galois connection between the powerdomain of* pairs *of $\mathcal{LSL}$ memory states and the powerdomain of 3-valued logical structures. (See, e.g., the shape abstraction suggested in [RBR+05, Sec. 5.1].)*

Instrumentation Predicates As mentioned before, the *instrumentation principle* in [SRW02] ensures that it is possible to refine the abstraction by using *instrumentation predicates* that may provide additional information that might be lost under abstraction. In particular, it is possible to refine the abstraction of cutpoints by adding additional unary instrumentation predicates.

4.8.3.2 Abstract Operational Semantics

Because our framework is based on [SRW02], the actions we used to define the concrete operational semantics for program statements (as transformers of 2-valued structures) in Section 4.8.2, also define the corresponding

abstract semantics (as transformers of 3-valued structures). This abstract semantics is obtained by reinterpreting logical formulae using a 3-valued logic semantics and serves as the basis for an abstract interpretation.

We manually provide the update formulae of the instrumentation predicates (as done e.g., in [SRW02, LARSW00, RS01, Yah01]). Automatic derivation of update formulae for the instrumentation predicates [RSL03] is currently not implemented for the types of structure manipulation employed by the **clone** and **remove**.

The reader may wonder how can we update the information after a call given the fact that we allow arbitrary number of cutpoints. The truth is that the transitive closure formulae are evaluated to indefinite $1/2$ values whenever the cutpoint objects become summary nodes. Therefore, our analysis becomes imprecise in these cases. (We note, however, that it is possible to match cutpoints precisely when they are not summarized. In our experiments we employed this optimization.)

The soundness of our abstract semantics is guaranteed by the combination of the theorems in Section 4.5, and [SRW02]:

- In Section 4.5, it is shown that the $\mathcal{LSL}$ semantics is *observationally equivalent* to a standard store-based global-heap semantics.
- In [SRW02], it is shown that every program-analyzer which is an instance of their framework is sound with respect to the concrete semantics it is based on.

4.9 Interprocedural Functional Analysis via Tabulation of Abstract Local-Heaps

Our algorithm computes partial procedure summaries by tabulating input abstract memory-states to output abstract memory-states. (As in Section 3.8, the tabulation is restricted to abstract memory-states that occur in the *analyzed* program). However, the tabulated abstract memory-states (conservatively) represent local-heaps and uses *abstract cutpoints* to represent the sharing patterns of the local-heap with its local context. Therefore, the tabulated abstract states are independent of the context in which a procedure is invoked. As a result, the summary computed for a procedure could be used at different calling contexts and at different call-sites.

Our interprocedural tabulation algorithm is a variant of the IFDS-framework [RHS95] adapted to work with local-heaps. The tabulation algorithm is described in Appendix F. This is the same tabulation algorithm used in Section 3.8, with different instantiations of the call and return rules. (See Appendix F for details.)

> **Example 4.9.1** Figure 4.19 shows a partial tabulation of abstract local-heaps for the `splice` procedure of the running example. The figure shows 3 possible input states of the list pointed-to by `p` where the second list element is a cutpoint.

4.9.1 Prototype Implementation

We have implemented a prototype of our framework using TVLA [LAS00]. To translate Java programs and their specifications to TVP (TVLA input language) we have extended an existing Soot-based [VRCG+99] front-end for Java developed by R. Manevich.

We handle dynamic-dispatch by selectively propagating *3-valued* logical structure over an interprocedural edge according to the type of the receiver object (information maintained in our analysis, by instrumenting our semantics to also track the types of objects. In our implementation, we do not use set-union as join-operator. Instead, we use a more "aggressive" partial-join operation [MSRF04]. This operation exploits the fact that our abstract domain has a Hoare order and returns an upper approximation of the set-union operator.

Our framework allows control over the heap-abstraction and the cutpoint abstraction in the instantiated algorithms. We have instantiated the framework to produce a shape-analysis algorithm for analyzing general heap-manipulating programs. In our experiments we used a class-based abstraction for cutpoints, i.e., we do not distinguish between different cutpoints of the same type. This means that we lose precision when a procedure is invoked with two or more cutpoint objects of the same class.

We applied our framework to verify various correct list-manipulating programs and to verify client conformance with API specification. Our measurements were obtained on a machine with a 1.5 Ghz Pentium M processor and 1 Gb memory.

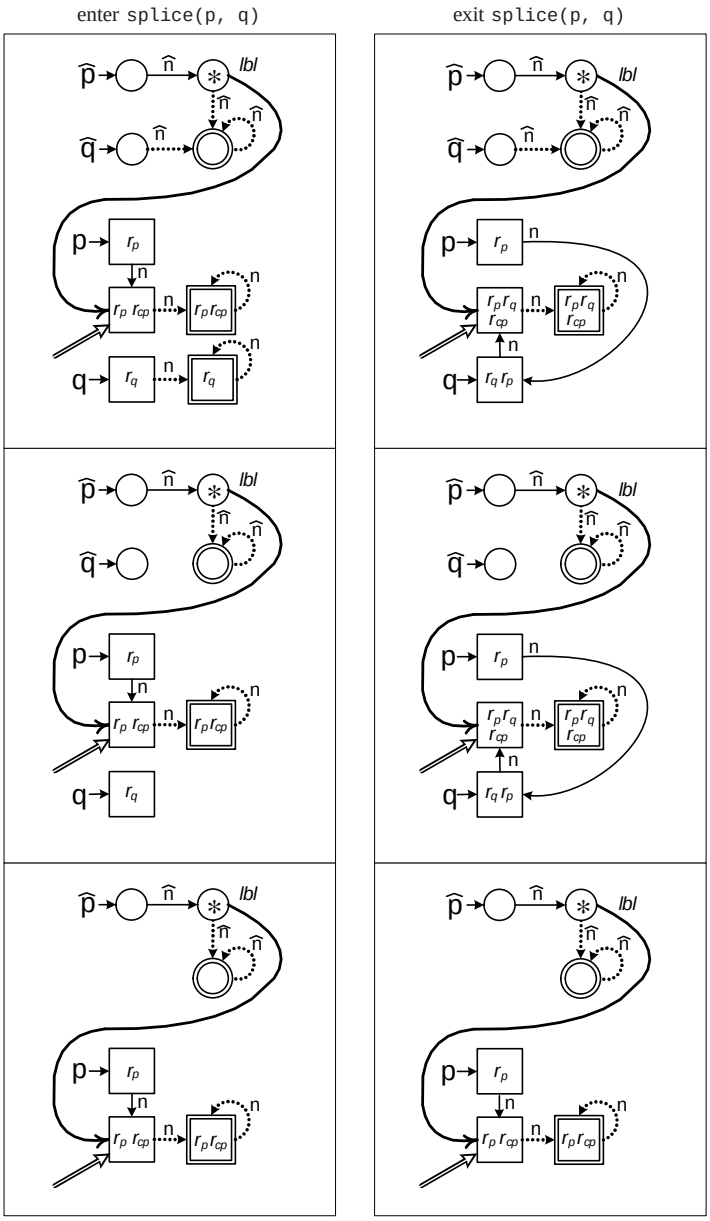


Figure 4.19: Partial tabulation of abstract states for the splice for an invocation with a single cutpoint. (In the third row the value of the formal parameter q is *null*, and thus it is not shown). For similar reasons, q is not shown in the third row, the sixth row, and the ninth row.

a. Program	Iterative		Recursive	
	Space (MB)	Time (Sec)	Space (MB)	Time (Sec)
create creates a list	19.7	10.9	19.3	9.3
find searches an element in a list	22.3	21.3	23.5	35.8
insert allocates and inserts an element into a sorted list	23.3	41.2	23.3	41.2
delete removes an element from a sorted list	23.2	42.0	24.8	45.3
append appends two lists	25.1	17.2	25.6	20.2
reverse destructive list-reversal	23.6	23.7	24.0	33.7
revApp reverses a list by appending its head to the reversed tail	26.0	45.7	26.5	46.8
merge merges two sorted lists	25.9	579.7	27.8	91.9
splice splices two lists	25.5	70.1	26.1	36.9
running the running example	27.7	160.0	28.3	45.7
b. Code fragment	**Code Inline**		**Proc. Calls**	
crt3 creates a list of length 3	22.3	5.4	22.0	6.4
crt3x3 creates 3 lists of length 3	50.7	27.0	26.2	9.2

Table 4.1: Analysis cost for list-manipulating programs.

Program	Line No.	Space (MB)	Time (Sec)	Rep. / Act. Errors
ISPath	71	24.9	1378.0	0/0
InputStream5	64	61.8	2484.4	0/0
InputStream5b	64	61.7	2550.5	1/1
JDBC Example fixed	153	191.0	25213.0	0/0
JDBC Example	149	191.9	25261.3	1/1

Table 4.2: Analysis results and cost for verifying client conformance with API specification.

Our analysis was able to verify that the list-manipulating programs do not perform null-dereferences and that the lists they manipulate are acyclic. Measurements of analysis cost for these programs are shown in Table 4.1(a). The table compares the cost of the analysis of a program which invokes an iterative procedure (first column) with the cost of the analysis of the same program, except that the invoked procedure is recursive[13] (second column). For these programs, we found that the cost of analyzing recursive procedures and iterative procedures is comparable in most cases. We note that our tests were of *client* programs and not a single procedure, i.e., in all tests, the program also allocates the list it manipulates. In addition, we compared the cost of the analysis of the three calls to `crt3` in our running example to the cost analyzing this code fragment when the body of `crt` is inlined in the `main` procedure (see Table 4.1(b)). We are encouraged by these results, as they indicate (at least in this simple example) that our analysis benefits from procedural abstraction.

We also applied our framework to verify correct usage of the `IO`-stream and the `JDBC` interfaces. In particular, we verified that closed files are not read (`IO`-streams) and that clients use database connections correctly (`JDBC`-client). The analysis cost and results are shown in Table 4.2. The column "Rep. / Act. Error" shows the number of reported errors, compared to the number of actual errors. `ISPath` is a simple correct program manipulating input streams. `InputStream5` is a program that stores input-streams at arbitrarily deep recursive data structures. `InputStream5b` is an erroneous version of `InputStream5` containing a single error. `JDBC Example` is an erroneous program which manipulates 5 database connection. `JDBC Example fixed` is a correct version of the last program. These programs were part of the benchmarks used in [YR04]. Our analysis verifies the correct use of the API with the same precision as the separation-based approach, but *without the need for specification*. However, the cost of the analysis was higher than the cost of the analysis in [YR04]. We attribute these to the fact that our heap analysis is more precise than the one used in [YR04].

[13] revApp is a recursive procedure. We analyzed it once one with an iterative append procedure and once with a recursive append.

4.10 Discussion: $\mathcal{LSL}^{\text{CPF}}$ vs. $\mathcal{LSL}$

In this section, we contrast $\mathcal{LSL}^{\text{CPF}}$ with $\mathcal{LSL}$ and discuss their advantages and the disadvantages with respect to shape analysis.

We note that both $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ are procedure local-heap storeless semantics: when a procedure is invoked, it operates only on a part of the heap, namely, the objects that are reachable from the procedure's actual parameters. Thus, both semantics (and their abstractions) have the advantage that their memory states do not represent parts of the heap which are not relevant to the current procedure. However, as a result, both semantics (and their abstractions) do not preserve properties of the parts of the heap which are irrelevant to the current procedure.

The two semantics differ in the access paths that they use: In $\mathcal{LSL}^{\text{CPF}}$, every dynamically allocated object o is represented by the set of pointer-access paths that *start at a local variable of the current procedure* and reach o. In contrast, in $\mathcal{LSL}$, every dynamically allocated object o is represented by the set of pointer-access paths that reach o and start at a local variable of the current procedure *or at one of its cutpoints*.

4.10.1 Advantages of $\mathcal{LSL}^{\text{CPF}}$

The main advantage of the approach taken by $\mathcal{LSL}^{\text{CPF}}$ is its simplicity which stems from the fact that it represents objects using standard access paths, i.e., access paths that start at variables. Furthermore, because every memory state is represented by access paths starting at the (local) variables of a single procedure, shape abstractions used for *intra*procedural shape analysis, e.g., [DRS00, MYRS05, DOY06, LAIS06], can be lifted to the interprocedural setting: Memory states can be abstracted using the shape abstraction used for the intraprocedural analyses. Furthermore, intraprocedural statements are handled as in the original analysis. To handle interprocedural call and return statements, the abstract domain needs to be augmented to support some additional operations: (i) carve out subheaps reachable from variables (i.e., restricting an abstract heap to the subset of its domain which is reachable from the actual parameters); (ii) combining disjoint subheaps; and (iii) modifying pointers (in the abstract state) that point to objects which are pointed to by variables (the actual parameters).

It is quite straightforward to add the aforementioned operations to the abstract domains of [DRS00, MYRS05, DOY06, LAIS06] because these domains record *reachability-from-variables* and *sharing* information, which is the information required to apply these operations. For a similar reason, it is quite straightforward to adapt abstract domains that were developed using the framework of [SRW02] which also record similar information.

4.10.2 Disadvantages of $\mathcal{LSL}^{\text{CPF}}$

The downside of the $\mathcal{LSL}^{\text{CPF}}$'s approach is that the memory state just after the call cannot always be defined in terms of the state prior to the call. The intuitive reason for this deficiency is that the description of an object may change due to destructive updates.

> **Example 4.10.1** Figure 3.6(a) depicts memory states that may occur, according to the $\mathcal{LSL}^{\text{CPF}}$ semantics, during the second cutpoint-free invocation of `splice` in the running example of Chapter 3. (Recall that this invocation is cutpoint-free). The figures depicts the memory states that might arise at the call-site ($\sigma^{c\,3.6}_{L_{cpf}}$), at the entry-site ($\sigma^{e\,3.6}_{L_{cpf}}$), at the exit-site ($\sigma^{x\,3.6}_{L_{cpf}}$), and at the return-site ($\sigma^{r\,3.6}_{L_{cpf}}$).
>
> Note that every object is represented in terms of access paths starting at local variables of procedure `splice`. As a result, the representation of the objects at the call site differs from their representation at the entry site. Nevertheless $\mathcal{LSL}^{\text{CPF}}$ can construct the memory state at the return site because (i) the representation of the irrelevant objects did not change, thus their representation can be taken, as is, from the call site; and (ii) by our simplifying assumptions, the formal variables could not have been assigned to; thus they point at the exit site to the same objects they pointed to at the entry site. Because the invocation is cutpoint-free, the object pointed to by parameters are the only objects that separate the local-heap of the callee from the rest of the heap. Thus, the semantics needs only to match the representation of the object pointed to by parameters at the exit site and at the return site. For example, the objects pointed to by `p` resp. `q` at the entry state remain to be pointed to by `p` resp. `q` at the exit state.
>
> Note that a shape abstraction that distinguish between objects that are reachable from different variables partitions the allocated objects in a way which allows to readily apply the additional operations

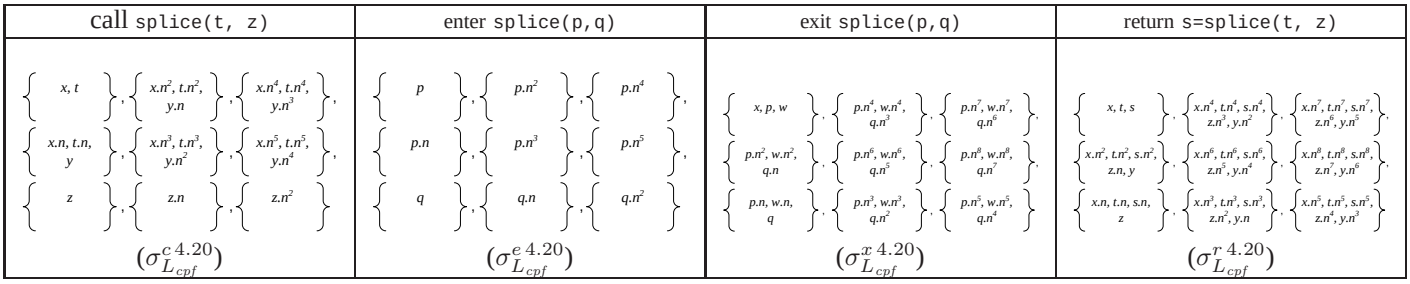


Figure 4.20: Memory states that *should have occurred* at the call-site, entry-site, exit-site, and return-site in the invocation `s = splice(t, z)` in the variant of the running example of Chapter 3 according to the $\mathcal{LSL}^{\text{CPF}}$ semantics *had it been allowed to execute*: $(\sigma^{c\,4.20}_{L_{cpf}})$ the call state, $(\sigma^{e\,4.20}_{L_{cpf}})$ the entry state, $(\sigma^{x\,4.20}_{L_{cpf}})$ the exit state, and $(\sigma^{r\,4.20}_{L_{cpf}})$ the return state.

required for handling *cutpoint-free* procedure invocations. For example, consider the shape abstraction described in Section 3.6 whose application for the invocation `s = splice(y, z);` is depicted in Figure 3.6(c).

In contrast, consider a non-cutpoint-free procedure invocation: Figure 4.20 depicts the $\mathcal{LSL}^{\text{CPF}}$ memory states that *should have occurred* during a non-cutpoint-free invocation *had it been allowed to execute*. Specifically, Figure 4.20 depicts the $\mathcal{LSL}^{\text{CPF}}$ memory states corresponding to the $\mathcal{LSB}$ memory states depicted in Figure 4.2(a). (These are memory states that may arise during the invocation `s = splice(t, z);` in the variant of the running example of Chapter 3).

Note that the object pointed to by `y` at the call state $(\sigma^{c\,4.20}_{L_{cpf}})$ has an entirely different representation at the entry state $(\sigma^{e\,4.20}_{L_{cpf}})$ and at the exit state $(\sigma^{x\,4.20}_{L_{cpf}})$: At the entry state it is represented by $\{p.n\}$ and at the exit state it is represented by $\{w.n.n, p.n.n, q.n\}$. Specifically, the memory state does not carry enough information for the semantics to determine that the two representations pertain to the same (cutpoint) object at different times.

$\mathcal{LSL}^{\text{CPF}}$ is able to use the standard notion of access paths because it avoids tracking the kind of temporal relationship which allows to match cutpoint objects at the entry state and at the exit state. Thus, $\mathcal{LSL}^{\text{CPF}}$ forbids cutpoints. As a result, any shape analysis based on $\mathcal{LSL}^{\text{CPF}}$ is required to detect t he possibility of a non cutpoint-free invocation. Specifically, the abstract domain should be able to answer queries regarding domination by variables. Such information is available in the aforementioned abstract domains.

4.10.3 Advantages of $\mathcal{LSL}$

The main advantage of the approach taken by $\mathcal{LSL}$ over the one taken by $\mathcal{LSL}^{\text{CPF}}$ is that it can handle arbitrary cutpoints. (See Example 4.4.4). This ability makes it observationally equivalent to the standard heap semantics. (See Theorem 4.5.3). (In contrast, $\mathcal{LSL}^{\text{CPF}}$ is observationally sound with respect to the standard heap semantics. See Theorem 3.5.3).

4.10.4 Disadvantages of $\mathcal{LSL}$

The main disadvantage of $\mathcal{LSL}$ is that it is rather complicated due to its use of non standard notion of *generalized* access paths. (See Definition 4.3.3). These access path are used to record temporal relationship which allows to match cutpoint objects at the entry state and at the exit state. As a result, when abstracting $\mathcal{LSL}$ one needs to devise abstractions for cutpoint-labels (the "sharing patterns"). In Section 4.7, we show how memory states of $\mathcal{LSL}$ can be represented using 2-valued logical structures. This allows to develop abstractions parametrically using canonic abstraction [SRW02], however, the designer of the analysis needs to specify the instrumentation predicates that determine the abstraction. Also, the designer of the analysis needs to specify the effect of program statements regarding cutpoint-specific information, e.g., reachability from cutpoints.

4.10.4.1 Abstraction of Cutpoints

We experimented with type-based abstraction, i.e., the abstraction may merge cutpoints of the same type. As a result the abstraction becomes imprecise (and inefficient) when there are more than one cutpoint of the same type.

A different approach which allows for multiple cutpoints for procedures with multiple formal arguments is to discriminate cutpoints reachable from different formal parameters. This will improve the precision of handling procedures that are passed multiple lists.

In [GBC06], a different approach is taken to abstract cutpoints: The analysis allows for procedure invocations as long as the number of cutpoints does not exceed a certain a priori fixed threshold. If the procedure has more cutpoints then allowed, the analysis does not represent the cutpoints at all. Instead, it verifies that all the references to the cutpoints are *dead*. (See Sections 6.3.3 and 7.2.2).

Chapter 5

Modular Shape Analysis for Dynamically Encapsulated Programs

This chapter presents a novel method for automatically verifying properties of heap manipulating programs in a modular *fashion: We consider a program to be a collection of modules and develop a shape (heap) analysis which treats each module separately. Our approach is focused on analyzing* dynamically encapsulated programs*: programs in which live references (i.e., used before set) between subheaps manipulated by different modules form a tree. Our (modular) analysis also conservatively verifies that the analyzed program is dynamically encapsulated.*

We formally define the set of dynamically encapsulated programs by means of a non-standard operational semantics ($\mathcal{DOS}$) which places certain restrictions on aliasing and sharing across modules. We use $\mathcal{DOS}$ as a basis for a concrete module semantics which assigns a program-independent meaning to every module. We develop conservative static analysis algorithms by abstract interpretation of the module semantics.

Our analysis is modular in the program code*: it analyzes each module separately and determines properties of the data structures manipulated by the module that hold in any dynamically encapsulated program. The analysis is also* modular in the heap*: when analyzing a module, the analysis explicitly represents only the data structure which are manipulated by that module.*

The material described in this chapter is largely based on the material that originally appeared in [RPHR$^+$07, RPHR$^+$06].

5.1 Introduction

This chapter presents a novel approach for modular shape analysis. In our approach, each module of a program is analyzed separately. Modular shape analysis is a particularly difficult problem due to aliasing: The behavior of a module can depend on the aliasing created by clients of the module and vice versa. Analyzing a module making worst-case assumptions about the aliasing created by clients (or vice versa) can complicate the analysis and lead to imprecise results.

Instead of analyzing arbitrary programs, we restrict our attention to certain "well-behaved" *dynamically encapsulated* programs, and describe an analysis that checks that the program is dynamically encapsulated and computes an over-approximation of the heap. The main idea behind our approach is to *assume and modularly verify* a (modularly-checkable) program-invariant concerning aliases of live intermodule references.

5.1.1 Main Results

The main contributions of this chapter can be summarized as follows:

(i) We introduce an interesting class of dynamically encapsulated programs;

(ii) We define a natural notion of module invariant for dynamically encapsulated programs;

(iii) We show how to utilize dynamic encapsulation to enable modular shape analysis; and
(iv) We present a modular shape analysis algorithm which (conservatively) verifies that a program is dynamically encapsulated and identifies its module invariants.

A distinguishing aspect of our work is that we integrate a shape analysis with encapsulation constraints. Our work presents a nice interplay between encapsulation and modular shape analysis: it uses dynamic encapsulation to enable modular shape analysis, and uses shape analysis to determine that the program is dynamically encapsulated. Thus, in this chapter, we discuss not only how shape analysis can be used to verify dynamic encapsulation, but also how dynamic encapsulation helps enabling modular shape analysis.

Partial Procedure Summaries vs. Full Procedure Summaries. The analysis algorithms presented in Chapters 3 and 4 are modular in the program's *heap*,[1] but not in the program's *code*, i.e., they analyze *whole* programs. In contrast, the analyses described in this chapter are modular in the program code, they analyze each *part* of the program separately.

In addition, the analyses described in Chapters 4 and 3 compute only *partial summaries* (see Section 3.8): they analyze procedures while considering only calling contexts that occur in the analyzed program. Thus, while it is possible to use the computed summaries across different programs, the analysis should always be ready to reanalyze the procedure in a calling context which was not already considered. In contrast, the approach presented in this chapter, computes *full* summaries (for dynamically-encapsulated programs), every procedure is analyzed in every possible calling context that might arise in any dynamically-encapsulated program. As a result, the analysis described in this chapter can establish properties that hold for the module in *any* dynamically-encapsulated program.

Specification. Our modular analysis requires some lightweight user specification. More specifically, the user is expected to define the partitioning of the program into modules and to provide information on the transfer of ownership. Informally, the ownership-transfer specification determines which references are not expected to be used (See Sections 1.3.3, 5.3.1.3, and 5.4.2).

Outline. The remainder of the chapter is organized as follows: Section 5.2 introduces our running example. Section 5.3 presents an informal overview of our approach. Section 5.4 formalizes our programming model and the specification language. Section 5.5 presents the $\mathcal{DOS}$ semantics and Section 5.6 investigates its properties. Section 5.7 formalizes the notion of conditional module invariants. Our modular shape analysis is presented in two main stages: In the first stage, Section 5.8 presents a concrete module semantics and in the second stage, Section 5.9 presents an abstract module semantics.

5.2 Motivating Example

Figure 5.1(b) shows the code of a module, m_{RP}, which serves as our running example. The code is written in a Java-like language. Module m_{RP} contains two classes: Class `R` is a class of resources to be used by clients of the module. A resource has a recursive field, `n`, which is used to link resources in an internal list.[2] Class `RPool` is a pool of resources which stores resources using their internal list. We assume that the `n`-field is read or written only by `RPool`'s methods: `acquire`, which gets a resource out of the pool, and `release`, which stores a resource in the pool. The `@transferred` annotations, used to specify the transfer of ownership, are explained in Sections 5.3.1.3 and 5.4.2. (An informal explanation of these annotations can be found in Section 1.3.3, which also uses the resource pool module as an example).

Typical properties we want to verify modularly are that for *any well behaved* program that uses m_{RP}, the methods of `RPool` never leak resources[3] and never give an acquired resource to a client before it is released.[4]

[1] See Sections 3.5 and 4.5.
[2] Analogous fields can be found, *e.g.*, in the Linux kernel timers [BC05].
[3] By never leaking a resource, we mean that once a resource is placed in a pool, it remains in the pool until it is acquired.
[4] Similarly, in the analysis of a client of m_{RP}, we would like to verify that the client does not use a dangling reference to a released resource. Our analysis can establish this property.

```
module mRP;

record  RPool := {
  ˇR rs;
}

@transferred: { e }
int  release(ˇR e) {
  e.n=this.rs;
  this.rs=e;
  ret = 0
}

@transferred: {  }
ˇR  acquire() {
  R r = this.rs;
  if (r!=null) {
      this.rs=r.n;
      r.n = null;
  }
  else {
      r = new R();
  }
  ret = r;
}


record  R := {
  ˇR n;
  ...
}
```

(a) The running example written in EAlgol$_M$.

```
package mRP;

public class RPool {
  private R rs;


  @transferred: { e }
  public void release(R e){
    e.n=this.rs;
    this.rs=e;
  }


  @transferred: { }
  public R acquire(){
    R r = this.rs;
    if (r!=null) {
      this.rs=r.n;
      r.n = null;
    }
    else {
      r = new R();
    }
    return r;
  }
}

public class R {
  R n;
  ...
}
```

(b) The running example written in JAVA.

Figure 5.1: The running example (a) in EAlgol$_M$ and (b) in JAVA.

Note that the aforementioned properties do not hold for arbitrary programs because of possible aliasing in the module induced by the client behavior: Consider an invocation of `p.release(r)` in a memory state in which `p` points to a non-empty resource pool.
- If `r` points to the head of a resource list containing more than one resource, then the tail of the list might be leaked.
- If, after being released into the pool that `p` points to, `r` is released into other pools, then these pools, along with the one pointed-to by `p` share (parts of) their resource lists. Note that after a shared resource is acquired from one pool, it can still be acquired from the other pools.
- Finally, if the resource that `r` points to is already in `p`'s pool, then `p`'s resource list becomes cyclic. A resource which is acquired from a pool whose list is cyclic, stays in the pool.[5]

5.3 Overview

In this section, we provide an informal overview of our approach for modular shape analysis.

5.3.1 $\mathcal{DOS}$: A Non-standard Dynamic Ownership Semantics

The basis for our approach is a *non-standard semantics* that captures the aliasing constraints mentioned above. In this chapter, a module is a collection of type-definitions and procedures, and a component is a subheap. Our semantics represents the heap as an evolving collection of (heap) *components*. Every component is comprised of objects whose *types are defined in the same module*. (We say that a component *belongs to* that module). Note that multiple components belonging to the same module may co-exist. References between components belonging to different *modules* are allowed, however, the *internal structure* of a *component* can be accessed or modified only by the (procedures in the) module to which it belongs.[6]

Components can be in two different states: *sealed* and *unsealed*. Sealed components represent encapsulated data returned by a module to its callers (and, hence, are expected to satisfy certain *module invariants*). In contrast, unsealed components are components that are currently being modified and may be in an unstable state.

At any point during program execution, the internal structure of only one component is "visible" and can be accessed or mutated, *i.e.*, only one unsealed component is "visible". We refer to this component as the *current component*. The only way a sealed component can become *unsealed* (permitting its internal structure to be examined and modified) is to pass it as a parameter of an appropriate intermodule procedure call so that the component becomes part of the current component for the called procedure. Our semantics requires that all parameters and the return value(s) of intermodule procedure calls must be sealed components. For simplicity, we do not consider primitive values here.

> **Example 5.3.1** Example 1.3.7 describes the component decomposition of a memory state, depicted in Figure 1.3(c), that might result in a program comprised of two modules: a resource manager and a resource pool. (In Example 1.3.7 we used the term *package* instead of a module).
>
> Assuming that the memory state depicted in Figure 1.3(c) occurs during an execution of one of the resource manager procedure, then the heap component containing the objects the resource manager and the two pool managers is unsealed, and can be mutated by the procedure. The other five components are sealed components. Their internal structure can be accessed only when they are passed as parameters to a procedure of the resource pool module.

5.3.1.1 Constraints

So far we have not really placed any constraints on the program. The above are standard "good modularity principles" and most programs will fit this model with minor adjustments. Before we describe the constraints we place on sharing across modules, we describe the two key challenges that motivate these constraints:

[5] Based on a bug we found in LEDA [MN99] while working on an earlier version of our approach [Rin01]. The bug (reported and fixed) was that concatenating a list to itself created a cycle.

[6] A module m can manipulate a component of a module m' by invoking an intermodule procedure call to a procedure of module m'.

Challenge I: How can we analyze a module M without using any information about the clients of M (*i.e.*, without using information about the usage context of M)?

Challenge II: When analyzing a client module C that makes use of another module M, how do we handle *intermodule* calls from C to M using only the analysis results for module M (*i.e.*, without analyzing module M again)?

We say that a component *owns* another component if it has a *live* reference (*i.e.*, used before set) to the other component. The most important constraint we place is that a component cannot be owned by two or more components. As a result, the heap (or the program state) may be seen as a tree of components. Informally, this ensures that distinct components do not share (live) state. Furthermore, we require that all references to a component from its owner have the same target object. We call this object the component's *header*.[7] We refer to a program which satisfies these constraints as a *dynamically encapsulated* program. Recall that our analysis also verifies that a program is *dynamically encapsulated*.

> **Example 5.3.2** The memory state depicted in Figure 1.3(c) is dynamically encapsulated: The inter-component references form a tree. (In Section 5.5, we show an example for a memory state which is dynamically encapsulated because the *live* inter-component references form a tree).

We require that the module dependency relation (see Section 5.4) be acyclic. This constraint simplifies our semantics (and analysis) as module reentrancy does not need to be considered: When a module is invoked *all* of its components are guaranteed to be sealed.

5.3.1.2 Benefits

The above constraints let us deal with the two challenges mentioned above in a tractable way. The restriction on sharing between components simplifies dealing with intermodule calls (Challenge II) as they cannot have unexpected side-effects: *e.g.*, an intermodule call on one component C_1 cannot affect the state of another component C_2 that is accessible to the caller. As for the first challenge, *we conservatively identify all possible input states for an intermodule call by iteratively identifying all possible sealed components that can be generated by a module*.

5.3.1.3 Specification

We now describe the extra specification a user must provide for the modular analysis. This specification consists of: (i) a *module specification* that partitions a program's types and procedures into modules; (ii) an annotation for every procedure that indicates for every parameter whether in an intermodule procedure invocation it is intended to be "transferred" to the callee or not; these annotations are only considered in intermodule procedure calls. A sealed component that is pointed to by a *transferred* parameter of an intermodule call cannot be subsequently used by the calling module (*e.g.*, to be passed as a parameter for a subsequent intermodule call). This constraint serves to directly enforce the requirement that the heap forms a tree of components.

> **Example 5.3.3** The resource parameter of procedure `release` is annotated as a transferred parameter. Consider a program which uses module m_{RP} and contains a procedure `goo`, defined in another module. Assume `goo` contains an invocation of procedure `release` using a pool parameter `p` and a resource parameter `r`. First, note that because `goo` is not part of module m_{RP}, both `p` and `r` point to sealed components whose content is "invisible" from the point of view of `goo`. Second, note that the invocation of `release` also transfers ownership of the resource pointed to by `r` to the pool pointed to by `p`. Specifically, `goo` cannot continue to use any reference it has to the transferred resource (i.e., any alias of `r`). However, `goo` still owns the pool pointed to by `p`, and thus can continue to use it.

Given the above specification, our modular analysis can automatically detect the boundaries of the heap-components and (conservatively) determine whether the program satisfies the constraints described above.

[7] Note the slight difference in terminology: In ownership type systems, owners are objects and do not belong to their ownership contexts. In our approach, components are the owners; the component header belongs to the component that is dominated by the header.

5.3.2 Conditional Module Invariants

Module invariants are properties of the module that hold in any program that respects the specification of the module. (I.e., these invariants hold in any program in which the procedures of the module are invoked only when their preconditions hold.) Our modular analysis conservatively identifies *module invariants* that hold in any program when it is executed according to the $\mathcal{DOS}$ *semantics*. However, when the program is executed according to the *standard semantics*, the determined invariants are guaranteed to hold only when the program is *dynamically encapsulated*. (In particular, it can be the case that these invariants may *not* hold in a *non*-dynamically encapsulated program, although this program respects the specification of the module.) To emphasize that according to the standard semantics the determined invariants are guaranteed to hold only for *dynamically encapsulated* programs, we refer to them as *conditional module invariants*. (See Section 5.7). We remind the reader that our analysis verifies that a program is dynamically encapsulated in a *modular* fashion. (See Sections 5.3.4 and 5.9).

We distinguish between two types of module invariants, *external conditional module invariants* and *internal conditional module invariants*:

- An *(external) conditional module invariant* of a module m (in *dynamically encapsulated* programs) is a property that holds for all the components that belong to m when they *are not* being used (*i.e.*, for sealed components).
- *Internal conditional module invariant* describes properties of the parts of the memory manipulated by module m when they *are* being used, i.e., properties of unsealed components. (Thus, we also refer to internal conditional module invariant as *conditional module implementation invariants*). These invariants are relational: they conservatively record for every procedure p the input/output relation between the parts of the memory manipulated by p's module when p is called from another module.

5.3.3 Concrete Module Semantics

We use the $\mathcal{DOS}$ semantics as a basis for a module semantics which assigns a program-independent meaning for every module. (Our modular analysis is obtained as an abstract interpretation of the module semantics). The module semantics is obtained by applying a novel *trimming abstraction* which abstracts away the contents of sealed components when analyzing a module. Loosely speaking, only the heap structure of the current component, and the aliasing relationships between intermodule references leaving the current component, are tracked. (Technically, the trimming abstraction is achieved by first applying the componentized heap abstraction, and then abstracting away the information regarding the contents of sealed components).

The module semantics is defined as a least fixpoint solution which is associated with every module. The main novel aspect of the module semantics is that it uses the constraints imposed by the $\mathcal{DOS}$ semantics to anticipate all possible calling contexts to the module.

5.3.4 Abstract Module Semantics

Our modular analysis is obtained as an abstract interpretation of the module semantics. We obtain an abstract module semantics by applying a *bounded* conservative abstraction of trimmed memory states. Rather than providing a new intraprocedural abstraction, we show how to *lift* existing *intra*procedural shape analyses, *e.g.*, [MYRS05, DOY06, LAIS06], to obtain a modular shape abstraction (see Section 5.9). Our analysis is parametric in the abstraction of trimmed memory states and can use different (bounded) abstractions when analyzing different modules.

5.3.4.1 Modular Shape Analysis for Dynamically Encapsulated Programs

Our static analysis is conducted in an assume-guarantee manner allowing each module to be analyzed separately. The analysis, computes a conservative representation of every possible sealed components of the analyzed module in dynamically encapsulated programs. This process identifies *conditional* structural invariants of the sealed components of the analyzed module, *i.e.*, it infers module invariants for dynamically encapsulated programs.

Given a module, and the user specification for the other modules it uses, our analysis tries to verify that the given module is "well-behaved". If this verification is unsuccessful, the analysis gives up and reports that the module may not adhere to our constraints. Otherwise, the analysis computes invariants of the given module that

hold in any "well-behaved" program containing the module. A program comprised only of successfully verified modules is guaranteed to be "well-behaved".

5.4 Program Model and Specification Language

In this section, we introduce EAlgol$_M$, a simple imperative procedural object-based (*i.e.*, without subtyping) language. EAlgol$_M$ is an extension of EAlgol, introduced in Section 2.1, with a module system. The syntax of EAlgol$_M$ is similar to that of EAlgol, defined in Figure 2.1, except that the grammatical rule for programs is defined as below:

$$P \in \mathit{prog} ::= \overline{\texttt{module}\ m;\ \overline{\mathit{rcdecl}}\ \overline{\mathit{prdecl}}}$$

Programs in EAlgol$_M$ consist of a collection of modules. Every module is a collection of procedures and user-defined type definitions. One module includes a `main` procedure, from which the execution of the program starts.

Example 5.4.1 Figure 5.1(a) shows the running example written in EAlgol$_M$.

Syntactic Domains. In addition to the semantic domains of EAlgol, defined in Section 2.1, we assume the syntactic domain of $m \in \mathcal{M}$ of module identifiers. As in EAlgol, we assume that variables, fields, procedures, types, program-labels and *modules* have unique identifiers in every program. We denote the type of a variable `x` resp. field `f` by $t(\texttt{x})$ resp. $t(\texttt{f})$.

Modules. We denote the module that a procedure p belongs to by $m(p)$ and the module that a type identifier T belongs to by $m(T)$.

We say that module m_1 *depends* on module m_2 if $m_1 \neq m_2$ and one of the following holds: (i) a procedure of m_1 invokes a procedure of m_2; (ii) a procedure of m_1 has a local variable whose type belongs to m_2; or (iii) a type of m_1 has a field whose type belongs to m_2.

Procedures. A procedure p has local variables (V_p) and formal parameters (F_p), which are considered to be local variables, *i.e.*, $F_p \subseteq V_p$. Only local variables are allowed. A procedure returns a value by assigning it to a designated variable `ret`. Parameters are passed by value.

5.4.1 Simplifying Assumptions

We assume that procedure invocations should be *cutpoint-free* (see Section 3.3). We explain the reasons for this assumption, and a possible relaxation, in Section 5.5.2.2.

To simplify the presentation, we make the following assumptions, in addition to the ones stated in Section 2.1:

(a) Objects of type T can be allocated and references to such objects can be *used as l-values* by a procedure p only if $m(p) = m(T)$;

(b) Actual parameters to an intermodule procedure call should not be aliased and should point to a component owned by the caller. In addition, we assume that parameters for inter-module procedure calls should have a non-*null* value.[8]

(c) The caller always becomes the owner of the return value of an intermodule procedure call.

5.4.2 Specification Language

We expect to be given a partitioning of the program types and procedures into modules.

Every procedure should have an ownership transfer specification given by a set $F_p^t \subseteq F_p$ of *transferred (formal) parameters*. (A formal parameter is a transferred parameter if it points to a transferred component in an intermodule call.) For example, `e` is `release`'s only transferred parameter, and `acquire` has none.

[8] This assumption simplifies the analysis by allowing us to avoid considering the different cases where different parameters have a *null*-value and non-*null* value when we determine the possible input states for intermodule procedure calls. See Section 5.8.4.

l	$\in$	Loc	Locations
v	$\in$	$Val = Loc \cup \{null\} \cup \{\ominus\}$	Values
ρ	$\in$	$\mathcal{E} = \mathcal{V} \hookrightarrow Val$	Environments
h	$\in$	$\mathcal{H} = Loc \hookrightarrow \mathcal{F} \hookrightarrow Val$	Heaps
t	$\in$	$\mathcal{T} = Loc \hookrightarrow \mathcal{T}$	Type maps
σ	$\in$	$\Sigma_D = \mathcal{E} \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M}$	Memory states

Figure 5.2: Semantic domains of the $\mathcal{DOS}$ semantics.

We define the set $F_p^{nt} = F_p \setminus F_p^t$ to be the set of non-transferred (formal) parameters of procedure p. For simplicity, we assume $\texttt{ret} \in F_p^{nt}$ in case the procedure returns a value. (Recall that by our simplifying assumptions, the caller always becomes the owner of the return value). For example, $F_{release}^{nt} = \{\texttt{this}\}$ and $F_{acquire}^{nt} = \{\texttt{this}, \texttt{ret}\}$.

5.5 $\mathcal{DOS}$: The Concrete Dynamic-Ownership Semantics

In this section, we define $\mathcal{DOS}$, a non-standard semantics which checks whether a program executes in conformance with the constraints imposed by the dynamic encapsulation model. ($\mathcal{DOS}$ stands for *dynamic-ownership semantics*).

$\mathcal{DOS}$ is a *store-based* semantics (see, *e.g.*, [MS77, NNH99, Rey02]). A traditional aspect of a store-based semantics is that a memory state represents a heap comprised of all the allocated objects. $\mathcal{DOS}$, on the other hand, is a *local-heap* store-based semantics (see Section 2.3): A memory state which occurs during the execution of a procedure does not represent objects which, at the time of the invocation, were not reachable from the actual parameters.

$\mathcal{DOS}$ is a small-step operational semantics [Plo81]. Instead of encoding a stack of activation records inside the memory state, as traditionally done, $\mathcal{DOS}$ maintains a *stack of program states*: Every program state contains a program point and a memory state. The program state of the *current procedure* is stored at the top of the stack, and it is the only one which can be manipulated by intraprocedural statements. When a procedure is invoked, the *entry memory state* of the callee is computed by a $Call$ operation according to the caller's current memory state, and pushed into the stack. When a procedure returns, the stack is popped, and the caller's *return memory state* is updated using a Ret operation according to its memory state before the invocation (the *call memory state*) and the callee's (popped) *exit memory state*.[9]

The use of a stack of program states allows us to represent in every memory state the (values of) local variables and the local-heap of just one procedure. An execution trace of a program P always begins with P's `main` procedure starts executing on an *initial memory state* in which all variables have a *null* value and the heap is empty. We say that a memory state is *reachable* in a program P if it occurs as the current memory state in an execution trace of P.

For a formal definition of the notions of *stacks of program states* and of *reachable memory states*, see Appendix E.1.

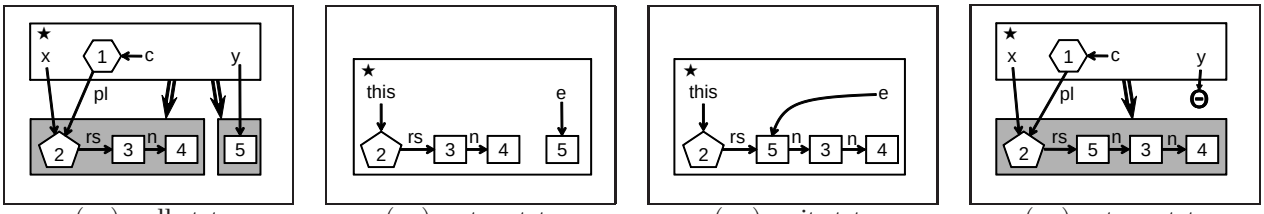
5.5.1 Memory States

Figure 5.2 defines the concrete semantic domains in $\mathcal{DOS}$ and the meta-variables ranging over them. We assume Loc to be an unbounded set of locations. A value $v \in Val$ is either a location, *null*, or $\ominus$, the *inaccessible value* used to represent references which should not be accessed.

A memory state in the $\mathcal{DOS}$ semantics is a 5-tuple $\sigma = \langle \rho, L, h, t, m \rangle$:

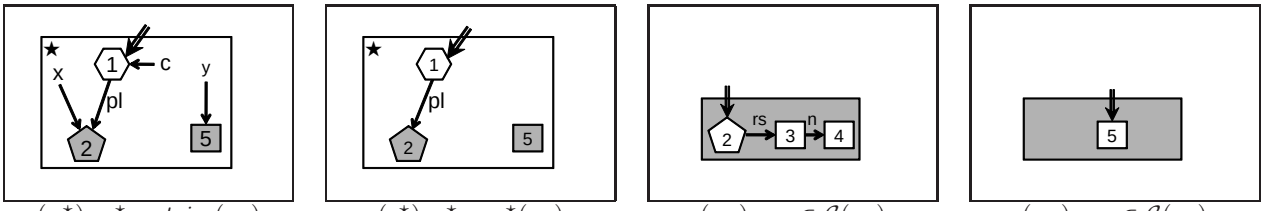
- The first three components comprise, essentially, a 2-level store, as introduced in Sections 2.2 and 2.3:
  - $\rho \in \mathcal{E}$ is an environment assigning values for the variables of the *current* procedure.
  - $L \subset Loc$ contains the locations of allocated objects. (An object is identified by its location. We interchangeably use the terms object and location.)
  - $h \in \mathcal{H}$ assigns values to fields of allocated objects.

[9] We note that our idea of using stacks of program states in the *concrete* semantics is heavily influenced by the formulation of an *interprocedural analysis* using a stack of *abstract* memory states in [KS92].



(σ_c) call-state (σ_e) entry-state (σ_x) exit-state (σ_r) return-state

(a) $\mathcal{DOS}$ memory states occurring in an invocation of `x.release(y)` on σ_c.



$(\sigma^\star)$ $\sigma^\star = trim(\sigma_c)$ $(c^\star)$ $c^\star = c^\star(\sigma_c)$ (c_P) $c_P \in \mathcal{C}(\sigma_c)$ (c_R) $c_R \in \mathcal{C}(\sigma_c)$

(b) $(c^\star, c_P, c_R)$: The implicit components of σ_c. $(\sigma^\star)$: The trimmed memory state induced by σ_c.

Figure 5.3: $\mathcal{DOS}$ memory states occurring in an invocation of `x.release(y)` on σ_c and the component decomposition of σ_c.

- $t \in \mathcal{TM}$ maps every allocated object to the type-identifier of its (immutable) type. Implicitly, t associates every allocated location to a module: The module that a location $l \in L$ *belongs to in memory state* σ, denoted by $m_\sigma(l)$, is $m(t(l))$. (When clear from context, we omit the σ subscript.)
- $m \in \mathcal{M}$ is the module of the current procedure. We refer to m as the *current module* of σ.

For convenience, we define selectors for memory states. Given a $\mathcal{DOS}$ memory state $\sigma = \langle \rho, L, h, t, m \rangle$, we define $\rho(\sigma) = \rho$, $L(\sigma) = L$, $h(\sigma) = h$, $t(\sigma) = t$, and $m(\sigma) = m$.

Note that in $\mathcal{DOS}$, reachability, and thus domination,[10] are defined with respect to the *accessible heap, i.e.*, $\ominus$-valued references do not lead to any object.

Example 5.5.1 Figure 5.3 (σ_c) depicts a possible $\mathcal{DOS}$ memory state that may arise in the execution of a program using the module m_{RP}. The state contains a *client* object (shown as an hexagon) pointed-to by variable `c` and having a `pl`-field pointing to a resource pool (shown as a pentagon).

The resource pool, containing two resources (shown as squares) is also pointed-to by a variable `x`. In addition, a local variable `y` points to a resource outside the pool. (The numbers attached to nodes indicate the location of objects. The value of a (non-null) pointer variable is shown as an edge from a label consisting of the variable name to the object pointed-to by the variable. *null*-valued variables are not shown in the figure. The value of a (non-null) field f of an object is shown as an f-labeled edge emanating from the object. The lack of such an edge indicates that the field has a *null* value. Other graphical elements can be ignored for now.)

The states σ_c and σ_e (also shown in Figure 5.3), depict, respectively, the call- and the entry-memory states of an invocation of `x.release(y)` which we use as an example throughout this section. Note that σ_e represents only the values of the local variables of `release` and does not represent the (unreachable) client-object. Note that in the return memory state of the invocation, depicted in Figure 5.3 (σ_r), the reference `y` has the $\ominus$-value, and that the resource pool dominates the resources in its list. Note, in particular, that the return state `y` *does not* have the value that it had before the call. (In Example 5.5.6, we explain how this return state is computed by the semantics).

[10] An object l_2 is *reachable from* (resp. *connected to*) an object l_1 in a memory state σ if there is a directed (resp. undirected) path in the heap of σ from l_1 to l_2. An object l is *reachable* in σ if it is reachable from a location which is pointed-to by some variable. An object l is a *dominator* if every access path pointing to an object reachable from l, must traverse through l. (See Section E.2.1.)

5.5.1.1 Components

Intuitively, a component provides a partial view of a $\mathcal{DOS}$ memory state σ. A component of σ consists of a set of reachable objects in σ, which all belong to the same module, and records their types, their link structure, and their *spatial interface i.e.*, references to and from immediately connected objects and variables.

More formally, a component $c \in \mathcal{C} = 2^{Loc} \times 2^{Loc} \times 2^{Loc} \times \mathcal{H} \times \mathcal{T} \times \mathcal{M} \times \mathcal{M}$ is a 6-tuple. A *component* $c = \langle I, L, R, h, t, m \rangle$ is a *component of* a $\mathcal{DOS}$ memory state σ if the following holds:

(i) L, the set of c's *internal objects*, contains only reachable objects in σ.

(ii) $I \subseteq L$ and $R \subseteq Loc \setminus L$ constitute c's spatial interface:
- I records the *entry locations* into c. An object inside c is an *entry location* if it is pointed-to by a variable or by a field of a *reachable* object outside c.
- R is c's *rim*. An object outside c is in c's rim if it is pointed-to by a field of an object inside c.

(iii) h defines the values of fields for objects inside c. We refer to a field pointing to an internal object as an *intra*-component reference. We refer to a field pointing to a rim object as an *inter*-component reference. h should be the restriction of σ's heap on L.

(iv) t defines the types of the objects inside c and in its rim. t should be the restriction of σ's type map on $L \cup R$.

(v) m is c's *component module*. We say that component c belongs to m. The type of every object inside c must belong to m. (If L is empty then m must be the current module of σ.) Note that a component c records (among other things) all the aliasing information available in σ pertaining to fields of c's internal objects.

(vi) For reasons explained below, we treat a variable pointing to a location outside the current component as an inter-component reference leaving the current component, and add that location to its rim (and relax the definition of a component accordingly).

For a formal definition of components, see Definition E.2.6.

> **Example 5.5.2** The call-memory state $\sigma_c = \langle \rho_c, L_c, h_c, t_c, m_c \rangle$, depicted in Figure 5.3, is comprised of three components. A rectangular frame encompasses the internal objects of every component. The current component, marked with a star, belongs to m_c, the client's module. The sealed components, drawn shaded, belong to module m_{RP}.
>
> Figure 5.3 ($c^\star$) depicts $c^\star = \langle I^\star, L^\star, R^\star, h^\star, t^\star, m_c \rangle$, the current component of the call-memory state, σ_c, separately from σ_c. The client-object is the only object inside $c^\star$. It is also an entry location, *i.e.*, $I^\star = L^\star = \{1\}$. An entry location is drawn with a double-line arrow pointing to it. The resource pool and the resource are rim objects, *i.e.*, $R^\star = \{2, 5\}$. (Note that location 5 is in the rim of $c^\star$ because of the relaxation mentioned in point (vi) above.) Rim objects are drawn opaque. The `pl`-labeled edge depicts the only (inter-component) reference in $c^\star$. Note that $h^\star = h_c|_{\{1\}}$ and $t^\star = t_c|_{\{1,2,5\}}$.
>
> Figure 5.3 (c_P) and (c_R) depict the sealed components of the call-memory state, σ_c, separately from σ_c.

The types of the reachable objects in a memory state σ induce its (unique) *implicit component decomposition*:

(i) a single *implicit current component*, denoted by $c^\star(\sigma)$, containing all the *reachable* objects in σ that belong to σ's current module and

(ii) a set of *implicit sealed components*, denoted by $\mathcal{C}(\sigma)$, containing (disjoint subsets of) all the *other* reachable objects. Two objects *reside within* the same implicit sealed component if they belong to the same module $m_s \neq m(\sigma)$ and are connected in σ's heap via an *undirected heap path* which only goes through objects that belong to module m_s.

The component decomposition of a memory state σ induces an *implicit component (directed) graph*. The nodes of the graph are the implicit components of σ. The graph has an edge from c_1 to c_2 if there is a rim object in c_1 which is an entry location in c_2, *i.e.*, if there is a reference from an object in c_1 to an object in c_2. For simplicity, we assume that the graph is connected, and treat local variables in a way that ensures that.

> **Example 5.5.3** Component $c^\star$, c_P, and c_R are the implicit components of σ_c, from Figure 5.3. I.e., $c^\star = c^\star(\sigma_c)$ and $\{c_P, c_R\} = \mathcal{C}(\sigma_c)$. Double-line arrows depict the edges of the component graph. This graph is connected because $c^\star$'s rim contains the resource pointed-to by `y`.

From now on, whenever we refer to a component of a memory state σ, we mean an implicit component of σ, and use the term *implicit component* only for emphasis. (For a formal definition of component graphs, see Definition E.2.9.)

5.5.1.2 Dynamically Encapsulated Memory States

We define the constraints imposed on memory states by the dynamic encapsulation model by placing certain restrictions on the allowed implicit components and induced implicit component graphs.

Definition 5.5.4 (Dynamic encapsulation) *A $\mathcal{DOS}$ memory state $\sigma \in \Sigma_D$ is said to be **dynamically encapsulated**, if (i) the implicit component graph of σ is a directed tree and (ii) every (implicit) sealed component in σ has exactly one entry location.*

We refer to the parent (resp. child) of a component c in the component tree as the *owner* of c (resp. a subcomponent of c). We refer to the single entry location of a sealed component c in a dynamically encapsulated memory state σ as c's *header*, and denote it by $hdr(c)$. We denote the module of a component c by $m(c)$.

Invariant 1 *The following properties hold in every dynamically encapsulated $\mathcal{DOS}$ memory state $\sigma \in \Sigma_D$:*

(i) A local variable can only point to a location inside $c^\star(\sigma)$, the current component of σ, or to the header of one of $c^\star(\sigma)$'s subcomponents.
(ii) For every component, every rim object is the header of a sealed component of σ.
(iii) A field of an object in a component of σ can only point to an object inside c, or to the header of one of c's subcomponents.
(iv) All the objects in a sealed component are reachable from the component's header.
(v) A header dominates its reachable heap.[10]
(vi) Every reachable object is inside exactly one component.
(vii) All the locations in one component are of the same module. The locations in the current component belong to the current module of σ.
(viii) If $c_1 \in \mathcal{C}(\sigma)$ owns $c_2 \in \mathcal{C}(\sigma)$ then $m(c_1)$ depends *on $m(c_2)$.*

$\mathcal{DOS}$ preserves dynamic encapsulation. Furthermore, Invariant 1 holds in every memory state which arises during any execution of any program according to the $\mathcal{DOS}$ semantics. Thus, from now on, whenever we refer to a $\mathcal{DOS}$ memory state, we mean a *dynamically encapsulated* $\mathcal{DOS}$ memory state. As a consequence of our simplifying assumptions and the acyclicity of the module dependency relation, the following holds for every $\mathcal{DOS}$ memory state σ:

(i) The internal objects of $c^\star(\sigma)$ are exactly those that the current procedure can manipulate without an (indirect) intermodule procedure call.
(ii) The rim of $c^\star(\sigma)$ contains all the objects which the current procedure can pass as parameters to an intermodule procedure call.

5.5.2 Operational Semantics

In this section, we define $\mathcal{DOS}$'s operational semantics. Formally, the semantics is specified as a relation between stacks of program states. Effectively, intraprocedural program statements and procedure call statements can affect only the top most memory state in the stack and interprocedural return statements can affect only the two top-most memory states in the stack. Thus, to define the meaning of an intraprocedural statement st, it suffices to specify the relation between the pre-state and the post-state

$$[\![st]\!] \quad \subseteq \quad \Sigma_D \times \Sigma_D \ ,$$

to specify the meaning of a procedure call statement, it suffices to specify the relation between the call-state and the entry-state

$$[\![Call_{y=p(x_1,\ldots,x_k)}]\!] \quad \subseteq \quad \Sigma_D \times \Sigma_D$$

and to specify the meaning of a procedure return statement, it suffices to specify the relation between the call- and -exit- states and the return-state

$$[\![Ret_{y=p(x_1,\ldots,x_k)}]\!] \quad \subseteq \quad (\Sigma_D \times \Sigma_D) \times \Sigma_D \ .$$

(For a more formal explanation of the lifting of process, see Section E.1.)

$\langle \mathtt{x} = \mathtt{null}, \sigma\rangle \overset{D}{\rightsquigarrow} \langle \rho[x \mapsto \mathit{null}], L, h, t, m\rangle$		
$\langle \mathtt{x} = \mathtt{y}, \sigma\rangle \overset{D}{\rightsquigarrow} \langle \rho[x \mapsto \rho(y)], L, h, t, m\rangle$	ACC	$\rho(y) \neq \ominus$
$\langle \mathtt{x} = \mathtt{y.f}, \sigma\rangle \overset{D}{\rightsquigarrow} \langle \rho[x \mapsto h(\rho(y), f)], L, h, t, m\rangle$	ACC	$\rho(y) \neq \ominus$
	CUR	$m_\sigma(\rho(y)) = m$
$\langle \mathtt{y.f} = \mathtt{x}, \sigma\rangle \overset{D}{\rightsquigarrow} \langle \rho, L, h[(\rho(y), f) \mapsto \rho(x)], t, m\rangle$	ACC	$\rho(y) \neq \ominus$
	CUR	$m_\sigma(\rho(y)) = m$
	ACC	$\rho(x) \neq \ominus$
$\langle \mathtt{x} = \mathtt{alloc\ T}, \sigma\rangle \overset{D}{\rightsquigarrow}$	NEW	$l \in \mathit{Loc} \setminus L$
$\langle \rho[x \mapsto l], L \cup \{l\}, h[l \mapsto I], t[l \mapsto T], m\rangle$	TYP	$m(T) = m$
$\langle \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}), \sigma\rangle \overset{D}{\rightsquigarrow} \sigma$	CMP	$\rho(x) \bowtie \rho(y)$
	ACC	$\rho(x) \neq \ominus,\ \rho(y) \neq \ominus$

Figure 5.4: Axioms for intraprocedural statements. $\sigma = \langle \rho, L, h, t, m\rangle$. The I function nullifies the fields of a newly allocated location. $\bowtie$ stands for either $=$ or $\neq$. When convenient, we sometimes treat h as an uncurried function, i.e., as a function from $\mathit{Loc} \times \mathcal{F}$ to Val. (`assume` statements are used to implement conditionals).

5.5.2.1 Intraprocedural Statements

Figure 5.4 defines the axioms for intraprocedural statements. The meaning of intraprocedural statements is described by a transition relation $\overset{D}{\rightsquigarrow} \subseteq (\Sigma_D \times \mathit{stms}) \times \Sigma_D$.

Essentially, intraprocedural statements are handled as usual in a two-level store semantics for pointer programs. All other statements are handled in the standard way. (See Sections 2.2 and 2.3). The only unique aspect of $\mathcal{DOS}$ is that it aborts if an inaccessible-valued pointer is accessed. More specifically, the main difference between the semantics of intraprocedural statements in $\mathcal{DOS}$ and in the standard 2-level store semantics is in the (ACC) side-condition of the rules pertaining to the manipulation of pointer fields. In short, the side-conditions ensure that:

(ACC) inaccessible values are not accessed;
(CUR) only fields of objects in the current component can be manipulated; and
(TYP) only types of the current module can be instantiated.

5.5.2.2 Interprocedural Statements

$\mathcal{DOS}$ is a local-heap store-based semantics (see Section 2.3): when a procedure is invoked, it starts executing on an *input heap* containing only the set of *available objects for the invocation*. An object is *available for an invocation* if it is a *parameter object, i.e.,* pointed-to by an actual parameter, or if it is reachable from one. We refer to a component whose header is a parameter object as a *parameter component*.

A local-heap semantics and its abstractions benefit from not having to explicitly represent unavailable objects. However, in general, the semantics needs to take special care of *cutpoints*, available objects that are pointed-to by an access path which bypasses the parameters (see Definition 3.3.2). In this chapter, we do not wish to handle the problem of analyzing programs with an unbounded number of cutpoints, which we consider a separate research problem (see Section 7.2.2). Thus, for simplicity, we require that *intramodule* procedure calls should be *cutpoint-free* (see Section 3.3), i.e., the parameter objects should dominate the available objects for the invocation.[10] (In general, we can handle a *bounded* number of cutpoints.[11])

Figure 5.5 defines the meaning of the Call and Ret operations pertaining to an arbitrary procedure call $y = p(x_1, \ldots, x_k)$:

- The meaning of a call statement is described using a transition relation $\overset{D}{\rightsquigarrow} \subseteq (\Sigma_D \times \mathit{Call}_{y=p(x_1,\ldots,x_k)}) \times \Sigma_D$.

[11] We can treat a bounded number of cutpoints as additional parameters: Every procedure is modified to have k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, the (modified) *semantics* binds the additional parameters with references to the cutpoints. This is the essence of [GBC06]'s treatment of cutpoints.

- The meaning of a return statement is described using a transition relation $\overset{D}{\rightsquigarrow} \subseteq (\Sigma_D \times \Sigma_D \times Ret_{y=p(x_1,\ldots,x_k)}) \times \Sigma_D$.

5.5.2.3 Procedure Calls

The $Call$ operation computes the callee's *entry memory state* (σ_e). First, it checks whether the call satisfies our *simplifying* assumptions. In case of an intramodule procedure invocation, the caller's memory state (σ_c) is required to satisfy the domination condition (CPF) ensuring cutpoint-freedom. Intermodule procedure calls are invoked under even stricter conditions which are fundamental to our approach: Every parameter object must dominate the subheap reachable from it. This ensures that distinct components are unshared. However, *there is no need to check these conditions as they are invariants in our semantics*: Invariant 1(i,iv,v) ensures that every parameter object to an intermodule procedure call is a header which dominates its reachable heap. (Note that Invariant 1(iv) can be exploited to check whether an object is a dominator by *only* inspecting access paths traversing through its component.) Thus, only our simplifying assumptions pertaining to non-nullness (LOC) and non-aliasing of parameters (DIF) need to be checked.

The entry memory state is computed by binding the values of the formal parameters in the callee's environment to the values of the corresponding actual parameters; projecting the caller's heap and type map on the available objects for the invocation; and setting the module of the entry memory state to be the module of the invoked procedure.

Note that in intermodule procedure calls, the change of the current module implicitly changes the component tree: all the available objects for the invocation which belong to the callee's module constitute the callee's current component. By Invariant 1(vi,viii) and the acyclicity of the module dependency relation, these objects must come from parameter components.

> **Example 5.5.5** Figure 5.3 (σ_e) shows the entry memory state resulting from applying the $Call$ operation pertaining to the procedure call `x.release(y)` on the call memory state σ_c, also shown in Figure 5.3. All the objects in σ_e belong to m_{RP}, and thus, to its current component. Note that the latter is, essentially, a fusion of c_P and c_R, the sealed components in σ_c.

Note: The current component of a $\mathcal{DOS}$ memory state $\sigma \in \Sigma_D$ is the root of the component tree induced by the *local-heap* represented in σ. In a *global-heap*, this current component might have been one or more non-root subcomponents of a larger component-tree which is only partially visible to the current procedure. For example, the current component of the client procedure is not visible during the execution of `release`.

5.5.2.4 Procedure Returns

The caller's return memory state (σ_r) is computed by a Ret operation. When an *intermodule* procedure invocation returns, Ret first checks that in the exit memory state (σ_x) every non-transferred formal parameter points to an object (OWN) which dominates its reachable subheap (DOM). This ensures that returned components are disjoint and, in particular, that the procedure's execution respected its ownership transfer specification. (Here we exploit the simplifying assumption that formal parameters cannot be assigned to).

Ret updates the caller's memory state (which reflects the program's state at the time of the call) by carving out the input heap passed to the callee from the caller's heap and replacing it instead with the callee's (possibly) mutated heap. In $\mathcal{DOS}$, an object never changes its location and locations are never reallocated. Thus, any pointer to an available object in the caller's memory state (either by a field of an unavailable object or a variable) points after the replacement to an up-to-date version of the object.

Most importantly, the semantics ensures that any future attempt by the caller to access a transferred component is foiled: We say that a local variable of the caller is *dangling* if, at the time of the invocation, it points to (the header of) a component transferred to the callee. A pointer field of an object in the caller's memory state which was unavailable for the invocation is considered to be *dangling* under the same condition. The semantics enforces the transfer of ownership by *blocking*: assigning the special value $\ominus$ to every dangling reference in the caller's memory state. (Blocking also occurs when an *intra*module procedure invocation returns to propagate ownership transfers done by the callee.)

Note that cutpoint-freedom ensures that the only object that separate the callee's heap from the caller's heap are parameter objects. Thus, in particular, the only references that might be blocked point to parameter objects.

$\langle Call_{y=p(x_1,\ldots,x_k)}, \sigma_c\rangle \overset{D}{\rightsquigarrow} \sigma_e$
$\sigma_e = \langle \rho_e, L_c, h_c|_{L_{rel}}, t_c|_{L_{rel}}, m(p)\rangle$
$\rho_e = [z_i \mapsto \rho_c(x_i) \mid 1 \le i \le k]$
where: $L_{rel} = R_{h_c}(\{\rho_c(x_i) \in Loc \mid 1 \le i \le k\})$

$\langle Ret_{y=p(x_1,\ldots,x_k)}, \sigma_c, \sigma_x\rangle \overset{D}{\rightsquigarrow} \sigma_r$
$\sigma_r = \langle \rho_r, L_x, h_r, t_r, m_c\rangle$
$\rho_r = (block \circ \rho_c)[y \mapsto \rho_x(ret)]$
$h_r = (block \circ h_c|_{L_c \setminus L_{rel}}) \cup h_x$
$t_r = t_c|_{L_c \setminus L_{rel}} \cup t_x$
where: $L_{rel} = R_{h_c}(\{\rho_c(x_i) \in Loc \mid 1 \le i \le k\})$
$\rho_x^{\ominus} = \rho_x[z \mapsto \ominus \mid m_c \neq m(p), z \in F_p^t]$
$block = \lambda v \in Val. \begin{cases} \rho_x^{\ominus}(z_i) & v = \rho_c(x_i), 1 \le i \le k \\ v & \text{otherwise} \end{cases}$

CPF	$m_c = m(p) \Rightarrow D_{\rho_c,h_c}(dom(\rho_c), \{x_1,\ldots,x_k\})$
DIF	$m_c \neq m(p) \Rightarrow \forall 1 \le i < j \le k : \rho_c(x_i) \neq \rho_c(x_j)$
LOC	$\forall 1 \le i \le k : \rho_c(x_i) \in Loc$
OWN	$m_c \neq m(p) \Rightarrow \forall z \in F_p^{nt} : \rho_x(z) \in Loc$
DOM	$\forall z \in F_p^{nt} : D_{\rho_x^{\ominus},h_x}(F_p^{nt}, \{z\})$

Figure 5.5: $Call$ and Ret operations for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ assuming p's formal variables are $z_1, \ldots, z_k$. $\sigma_c = \langle \rho_c, L_c, h_c, t_c, m_c\rangle$. $\sigma_x = \langle \rho_x, L_x, h_x, t_x, m_x\rangle$. $F_p^{nt} = \{ret\} \cup (F_p \setminus F_p^t)$. Variable `ret` is used to communicate the return value. We use the following functions and relations: $R_h(L)$ computes the locations which are reachable in heap h from the set of locations L. The auxiliary relation $D_{\rho,h}(V_I, V_D)$ holds if the set of objects pointed-to by a variable in V_D, according to environment ρ, dominates the part of heap h reachable from them, with respect to the objects pointed-to by the variables in V_I.

When an intermodule call returns, and the current module changes, the component tree is changed too: The callee's current component may be split into different components whose headers are the parameter objects pointed-to by non-transferred parameters. These components may differ from the (input) parameter components.

Example 5.5.6 Figure 5.3 (σ_r) depicts the memory state resulting from applying the Ret operation pertaining to the procedure call `x.release(y)` on the memory state σ_c and σ_x, also shown in Figure 5.3. The insertion of the resource pointed-to by `y` at the call-site into the pool has (implicitly) fused the two m_{RP}-components. Note that according to the standard semantics, `y` should point to the first resource in the list. This would violate dynamic encapsulation. $\mathcal{DOS}$, however, utilizes the *ownership specification* to block `y` thus preserving dynamic encapsulation.

5.6 Properties of $\mathcal{DOS}$

In this section, we investigate the properties of the $\mathcal{DOS}$ semantics. More specifically, we show that $\mathcal{DOS}$ is observationally sound with respect to the standard semantics. Thus abstractions of $\mathcal{DOS}$ can be used to conservatively verify properties of programs with respect to the standard semantics. We also show that despite the fact that $\mathcal{DOS}$ is defined using a 2-level store, it has a storeless nature as it does not distinguish between isomorphic memory states.

5.6.1 Executions, Paths, and Reachable States

Before we review the properties of the $\mathcal{DOS}$ semantics, we introduce some standard notions regarding the (interprocedural) executions, paths, and reachable states of programs in $\mathcal{DOS}$. In this section, we provide an informal description of these notions, which are formally defined in Section E.1.4.

Stacks. Recall that $\mathcal{DOS}$ actually manipulates *stacks of program states*. A stack stk is a sequence of program states, i.e., a sequence[12] of pairs of a program point and a memory state. We denote the *height of stack* stk by $|stk|$. We denote the *top program state of stack* stk by $top(stk) \in PP \times \Sigma_D$.

[12] See Definition A.1.7.

Executions. An execution π in $\mathcal{DOS}$ is a sequence of *stacks*. We denote the *length of* π by $|\pi|$. We denote the *i-th stack that arises in an execution* π (where for $0 \leq i \leq |\pi|$) by $\pi(i)$. We refer to the top memory state in the first stack of program states of an execution π, as π's *initial memory state*, and denote it by $in(\pi)$. Similarly, we refer to the top memory state in the last stack of program states of an execution π, as π's *final memory state*, and denote it by $out(\pi)$.

Paths. We denote the sequence of program points induced by the program points at the top of every stack in an execution π by $path(\pi)$, i.e., $path(\pi) = [i \mapsto pp_i \mid \langle pp_i, \sigma_i \rangle = top(\pi(i))]$.

Feasible Executions. We say that an execution π is *feasible* for a program P is there is an execution of P starting from the initial memory state (see Section 5.5). We denote the set of feasible executions of P by Π^P.

Reachable Memory States. We say that an memory state $\sigma \in \Sigma_D$ is *reachable* by reachable execution $\pi \in \Pi^P$ of a program P, and denote it by $out(\pi)$, if $\langle pp, \sigma \rangle = top(\pi)$, for some program point pp.

Error state. Before continuing, we slightly modify the $\mathcal{DOS}$ semantics in the following way. $\mathcal{DOS}$ dictates when dynamic encapsulation is violated, and gets into an error state. We change this behavior by adding a specially designated error state, $\mathbb{E}^{DOS}$, and assuming that the semantics goes into this state when is detects a violation of the dynamic encapsulation restriction. Once the semantics reaches the error state, it keeps on propagating it, i.e., $\langle st, \mathbb{E}^{DOS} \rangle \xrightarrow{d} \mathbb{E}^{DOS}$ for every $st \in$ *stms*.

5.6.2 Observational Soundness

In this section, we formally define the notions of *observational soundness* between the $\mathcal{DOS}$ semantics and the standard heap semantics.

In $\mathcal{DOS}$, as well as in $\mathcal{GSB}$, the only means by which a program can observe a state are access paths. (See Definition 2.4.2). We say that two values are *comparable* in $\mathcal{DOS}$ if neither one is $\ominus$, the inaccessible value. We say that a memory state σ in the $\mathcal{DOS}$ semantics is *observationally sound* with respect to memory state s_G in the $\mathcal{GSB}$ memory state, if every pair of access paths that have comparable values in σ, has equal values in σ iff they have equal values in s_G.

By definition $\mathcal{DOS}$ *mimics* $\mathcal{GSB}$: Executing the same sequence of statements in the $\mathcal{DOS}$ semantics and in the standard semantics either results in a $\mathcal{DOS}$ memory state which is observationally sound with respect to the resulting standard memory state, or the $\mathcal{DOS}$ execution gets into an error state due to a constraint breach (detected by $\mathcal{DOS}$). A program is *dynamically encapsulated* if it does not have an execution which gets into an error state. (Note that the initial state of an execution in $\mathcal{DOS}$ is observationally sound with respect to its standard counterpart).

Our goal is to detect structural invariants that are true according to the *standard semantics*. $\mathcal{DOS}$ acts like the standard semantics as long as the program's execution satisfies certain constraints. $\mathcal{DOS}$ enforces these restrictions by blocking references that a program should not access. Similarly, our analysis reports an invariant concerning equality of access paths only when these access paths have comparable values.

An invariant concerning equality of access paths in $\mathcal{DOS}$ for a dynamically encapsulated program is also an invariant in the standard semantics. This makes abstract interpretation algorithms of $\mathcal{DOS}$ suitable, for example, for verifying data structure invariants, for detecting memory error violations, and for performing compile-time garbage collection.

Definition 5.6.1 (Value of access paths in the $\mathcal{DOS}$ semantics) *The value of an access path $\alpha = \langle x, \delta \rangle$ in state $\sigma = \langle \rho, L, h, t, m \rangle$ of the $\mathcal{DOS}$ semantics, denoted by $[\![\alpha]\!]_{\mathcal{DOS}}(\sigma)$, is defined to be $\hat{h}(\rho(x), \delta)$, where*

$$\hat{h}\colon \mathit{Val} \times \Delta \hookrightarrow \mathit{Val} \text{ such that}$$
$$\hat{h}(v, \delta) = \begin{cases} v & \text{if } \delta = \epsilon \quad \text{(note that } v \text{ might be } \ominus\text{)} \\ \hat{h}(h(v, f), \delta') & \text{if } \delta = f\delta',\ v \in \mathit{Loc} \\ \bot & \text{otherwise} \quad \text{(note that } v \text{ might be } \ominus\text{)} \end{cases}$$

Note that traversal of the inaccessible value is not defined.

Definition 5.6.2 (Comparable values) *A pair of values of the $\mathcal{DOS}$ semantics $v_1, v_2 \in Val$ is **comparable**, denoted by $v_1 \overset{?}{\bowtie} v_2$, $v_1 \neq \ominus$ and $v_2 \neq \ominus$.*

We define the notion of *observational soundness* between a $\mathcal{DOS}$ memory state σ and a standard 2-level store s_G as the preservations in s_G of all equalities and inequalities which hold in σ. Note that the preservation in the other direction is not required. Also note that an equality resp. inequality between access paths holds in σ only when the two access paths have comparable values. For simplicity, we define $[\![\texttt{null}]\!]_{\mathcal{DOS}}(\sigma) = [\![\texttt{null}]\!]_{\mathcal{GSB}}(\sigma) =$ *null*.

Definition 5.6.3 (Observational soundness) *The memory state $\sigma \in \Sigma_D$ is **observationally sound** with respect to memory state $s_G \in \mathcal{S}_G$, denoted by $s_G \leqq \sigma$, if for every $\alpha, \beta \in AccPath \cup \{\texttt{null}\}$ it holds that*

if $[\![\alpha]\!]_{\mathcal{DOS}} \overset{?}{\bowtie} [\![\beta]\!]_{\mathcal{DOS}}$ then $[\![\alpha]\!]_{\mathcal{DOS}}(\sigma) = [\![\beta]\!]_{\mathcal{DOS}}(\sigma) \Leftrightarrow [\![\alpha]\!]_{\mathcal{GSB}}(s_G) = [\![\beta]\!]_{\mathcal{GSB}}(s_G)$

We define the notion of observational soundness between two $\mathcal{DOS}$ memory states (resp. two standard memory states) in a similar manner.

Theorem 5.6.4 states that $\mathcal{DOS}$ (i) detects if an execution is dynamically-encapsulated, and (ii) for dynamically encapsulated executions, execution of statements preserves observational equivalence. The theorem follows from having the definition of the $\mathcal{DOS}$ semantics mimic a standard store-based semantics with some additional assignments of the inaccessible value at procedure returns. However, once this value is observed, the $\mathcal{DOS}$ semantics moves into an error state.

Before stating the theorem, we introduce the notion of *pairs of program states producible by a same level execution*, i.e., pairs of states that occur in the same invocation of a procedure in a particular execution: We say that a pair of program states $\langle\langle pp, \sigma\rangle, \langle pp', \sigma'\rangle\rangle$ is *producible by a same level execution* of P if there exists an reachable execution $\pi \in \Pi^P$ of P and $0 \leq i < j \leq |\pi|$ such that (i) $|\pi(i)| = |\pi(j)|$, (ii) $\langle pp, \sigma\rangle = \pi(i)$, (iii) $\langle pp', \sigma'\rangle = \pi(j)$, and (iv) $|\pi(i)| \leq |\pi(k)|$ for every $i \leq k \leq j$. (For a formal definition, see Section E.1.4).

Theorem 5.6.4 (Observational Soundness) *Let st be a statement and σ be a $\mathcal{DOS}$ memory state which is observationally sound with respect to a memory state s_G of the $\mathcal{GSB}$ semantics, i.e., $s_G \leqq \sigma$. If $\langle st, s_G\rangle \overset{GSB}{\rightsquigarrow} {s_G}'$ then there is a same-level execution of st in $\mathcal{DOS}$ starting from memory state σ ends in a memory state σ' such that ${s_G}' \leqq \sigma'$ or $\sigma' = \mathbb{E}^{DOS}$.*

The following lemma falls out from Theorem 5.6.4. The theorem states that for dynamically-encapsulated programs $\mathcal{DOS}$ can be used to: (i) verify data-structure invariants that are expressed by access-path equalities at a program point; and (ii) assert the absence of *null*-valued pointer dereferences. Formally, a property is an invariant at a (labeled) statement if it is satisfied in any memory-state that occurs just before the (labeled) statement is executed.

Lemma 5.6.5 *Let P be a dynamically encapsulated program. An invariant concerning equality of access paths in the $\mathcal{DOS}$ semantics is an invariant in $\mathcal{GSB}$.*

The following lemma states that for dynamically-encapsulated programs $\mathcal{DOS}$ can detect memory leaks[13] without investigating reachability from *roots* of pending access paths. A memory leak can occur only when a variable or a field is assigned `null`.

Lemma 5.6.6 *Let P be a dynamically encapsulated program. If a reference has the inaccessible value at a given program point in every execution of P, then this reference is not live (i.e., used before set) at that program point according to the standard semantics.*

[13]By a memory leak we mean an object that is not pointed-to by any access path; i.e., neither by an access path of the current call nor by one of a pending call.

5.6.3 Storelessness

We say that two $\mathcal{DOS}$ memory states are observationally equivalent if they are observationally sound with respect to each other. Note that, in particular, given two observationally equivalent $\mathcal{DOS}$ memory states, an access path has an undefined value in one state if and only if it has an undefined value in the other.

Definition 5.6.7 (Observational equivalence) *The $\mathcal{DOS}$ memory states $\sigma_1, \sigma_2 \in \Sigma_D$ are **observationally equivalent**, denoted by $\sigma_1 \lessapprox \sigma_2$, if $\sigma_1 \leqq \sigma_2$ and $\sigma_2 \leqq \sigma_1$.*

The following lemma shows that $\mathcal{DOS}$ is indifferent to location names. In this respect, $\mathcal{DOS}$ has the flavor of a storeless semantics.

Theorem 5.6.8 ($\mathcal{DOS}$ is indifferent to location names) *Let π_1, π_2 be executions of a program P according to the $\mathcal{DOS}$ semantics. If $|\pi_1(0)| = |\pi_2(0)| = 1$, $in(\pi_1) \lessapprox in(\pi_2)$ and $path(\pi_1) = path(\pi_2)$ then $out(\pi_1) \lessapprox out(\pi_2)$.*

The following lemma, shows that the $\mathcal{DOS}$ semantics does not depend on the immutability of locations of allocated objects. In particular, it shows that it is possible to replace the exit state of an invoked procedure with any observationally equivalent state. However, here we have to be a bit careful not to introduce clashes between location names in the call- and exit- states. Specifically, we need to make sure that memory locations which are used by the caller, but were irrelevant for the call were not allocated by the callee.

Definition 5.6.9 *Two $\mathcal{DOS}$ memory states σ_c and σ_x are **possible call- and exit- memory states for an arbitrary intramodule procedure invocation** $y = p(x_1, \ldots, x_k)$ if:*

(i) $L_c \subseteq L_x$ *and*

(ii) $(L_c \setminus L_{rel}) \cap L_x = \emptyset$*, were L_{rel} is as defined in Figure 5.5, i.e., the set of locations which are relevant to the invocation.*

Lemma 5.6.10 *Let σ^c, σ_1^x, and σ_2^x be $\mathcal{DOS}$ memory states such that (i) $\sigma_1^x \lessapprox \sigma_2^x$, (ii) the invocation $y = p(x_1, \ldots, x_k)$ is dynamically encapsulated in σ^c, and (iii) σ_c and σ_i^x (for $i = 1, 2$) are possible call- and return-memory states for this invocation.*

$$\langle \sigma^c, \sigma_1^x, Ret_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} \sigma_1^r \iff \langle \sigma^c, \sigma_2^x, Ret_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} \sigma_2^r$$

and $\sigma_1^r \lessapprox \sigma_2^r$.

Sketch of Proof: The core of the proof relies on the following observation: When a procedure returns, the *blocking* operation has two effects: (i) It *blocks* dangling reference to components whose ownership was transferred, and (ii) it *redirects* any dangling reference which is aliased with an actual parameter at the call site to the value of the corresponding formal parameter at the exit state. (In $\mathcal{DOS}$, the value of an accessible formal parameter is the same as that of its actual parameter, however $\mathcal{DOS}$ does not depend on this property).

5.7 Conditional Module Invariants

Our modular analysis identifies conservative *module invariants*. These invariants are true in *any* program according to the $\mathcal{DOS}$ semantics and in *any dynamically encapsulated program* according to the standard semantics. Thus, our module invariants are *conditional*, they are guaranteed to hold only for *dynamically encapsulated* programs.

We distinguish between two types of module invariants: *external conditional module invariants* and *internal conditional module invariants*. The external conditional module invariants of module m (which we usually refer to as the *module invariants of module* m) are invariants pertaining to sealed components of module m. The internal conditional module invariants of module m (which we also refer to as the *implementation invariants of module* m) are invariants pertaining to unsealed components of module m.[14]

[14] Recall that when a procedure of module m executes, the (*single*) component of module m is *unsealed*. In contrast, when a procedure of module m does not execute, all components of module m are *sealed*.

5.7.1 External Conditional Module Invariants

An *(external) conditional module invariant* of a module m (in *dynamically encapsulated* programs) is a property that holds for all the components that belong to m when they *are not* being used (*i.e.*, for sealed components).

More formally, the *module invariant of module m for type T*, denoted by $[\![Inv_m\ T]\!] \subseteq 2^{\mathcal{C}}$, is a set of sealed components of module m whose header is of type T: a sealed component c is in $[\![Inv_m\ T]\!]$ iff there exists a reachable $\mathcal{DOS}$ memory state σ in some program such that $c \in \mathcal{C}(\sigma)$.

Definition 5.7.1 (Conditional (External) Module Invariants) *The **conditional (external) module invariant of type T of module $\mathbf{m}$**, denoted by $[\![Inv_m\ T]\!] \subseteq 2^{\mathcal{C}}$, is the set containing every sealed component of module m whose header is of type T in any memory state that may arise during an execution of any program P that uses m.*

$$[\![Inv_m\ T]\!] = \left\{ c \in \mathcal{C}(\sigma) \;\middle|\; \sigma \in \Pi^P, m(\sigma) = m, t(hdr(c)) = T \right\} .$$

We say that a set S is a sound external module invariant of module $\mathbf{m}$ for a type T of module m *if* $[\![Inv_m\ T]\!] \subseteq S$.

> **Example 5.7.2** The module invariant of module m_{RP} for type `RPool` in our running example is the set containing all resource pools with a (possibly empty) *acyclic* finite list of resources.
>
> The module invariant of module m_{RP} for type `R` is the singleton set containing a single resource with a *nullified* `n`-field: An acquired resource always has a *null*-valued `n`-field and a released resource is inaccessible. (See also Example 1.3.9).

5.7.2 Internal Conditional Module Invariants

Internal conditional module invariants describe properties of the parts of the memory manipulated by module m when they *are* being used, i.e., properties of unsealed components. (Thus, we also refer to internal conditional module invariants as *conditional module implementation invariants*).

To formally define the domain of internal conditional module invariants, we introduce the notion of *trimmed memory states*.

5.7.2.1 Trimmed States

A *trimmed state* is a pair of an environment and the current component. Intuitively, a trimmed memory state can be thought of as a conservative approximation of a $\mathcal{DOS}$ memory state which abstracts away all the information regarding the shape of the sealed components and the structure of the component tree.

More technically, the **domain of trimmed states** is $\sigma^\star, \langle \rho, c^\star \rangle \in \Sigma^\star = \mathcal{E} \times \mathcal{C}$. A **trimmed state** $\sigma^\star = \langle \rho, c^\star \rangle = \langle \rho, \langle I, L, R, h, t, m \rangle \rangle \in \Sigma^\star$ is a pair of an environment and an unsealed component. The only locations that the environment may map variables to are the entry-locations of $c^\star$ and the locations of its rim-objects, i.e., $range(\rho) \subseteq I \cup R \cup \{\mathit{null}, \ominus\}$.

Remark 5.7.3 *Note that by Invariant 1(i), a local variable in a (dynamically encapsulated) memory state can point only to an object which is inside the current component or in its rim.*

> **Example 5.7.4** Figure 5.3 ($\sigma^\star$) depicts the trimmed memory state induced by the $\mathcal{DOS}$ memory state shown in Figure 5.3 (σ_c).

Given a $\mathcal{DOS}$ memory state $\sigma \in \Sigma_D$, we denote by $trim(\sigma) = \langle \rho, c^\star(\sigma) \rangle$ the *trimmed state induced by σ*. Similarly, given a set $\mathcal{DOS}$ memory state $\mathrm{S} \subseteq \Sigma_D$, we denote by $trim(\mathrm{S}) = \{trim(\sigma) \mid \sigma \in \mathrm{S}\}$ the *set of trimmed state induced by* S.

5.7.2.2 Relational Internal Conditional Module Invariants

Internal conditional module invariants at a program point pp in a procedure p of a module m (in *dynamically encapsulated* programs) is a relation that holds between the values of the environment and the current component at the entry to p and at pp.

Definition 5.7.5 (Internal conditional module invariants) *The **internal conditional module invariant at program point** pp **in procedure** p **of module** $\mathbf{m}$, denoted by $[\![Inv_m^{imp}\ pp]\!] \subseteq \Sigma^\star \times \Sigma^\star$, is the set of pairs of trimmed states such that $\langle \sigma_e^\star, \sigma^{\star\prime} \rangle \in [\![Inv_m^{imp}\ pp]\!]$ iff there exists a program P which uses m, and two $\mathcal{DOS}$ memory states σ_e and σ' such that*

(i) $\sigma_e = \langle \rho_e, L_e, h_e, t_e, m \rangle$ such that $\sigma_e^\star = trim(\sigma_e)$;
(ii) $\sigma' = \langle \rho', L', h', t', m \rangle$ such that $\sigma^{\star\prime} = trim(\sigma')$; and
(iii) $\langle \langle entry_p, \sigma_e \rangle, \langle pp', \sigma' \rangle \rangle$, where $entry_p$ is the entry site of procedure p, are producible by a same level execution of P.

We define *the program independent meaning of a procedure*, denoted by $[\![Inv_m^{imp}\ p]\!]$, to be the module implementation invariant at the exit-site of p.

Definition 5.7.6 (Program independent meaning of a procedure) *The **program independent meaning of a procedure** p **of module** m, denoted by $[\![Inv_m^{imp}\ p]\!]$, is $[\![Inv_m^{imp}\ p]\!] = [\![Inv_m^{imp}\ exit_p]\!]$, where $exit_p$ is the exit point of procedure p.*

We say that a set S of pairs of trimmed memory states of a procedure p is a *sound internal module invariant at program point pp in procedure p of module m* if $[\![Inv_m^{imp}\ pp]\!] \subseteq S$. We say that a set S is a *sound modular meaning of a procedure p* if $[\![Inv_m^{imp}\ p]\!] \subseteq S$.

5.8 Concrete Module Semantics

Our goal in this chapter is to define a modular analysis which (conservatively) computes (external and internal) module invariants. To achieve this goal, we first define a program-independent *concrete module semantics*. The concrete module semantics associates every module with its internal and external module invariants. These invariants are defined as the least fixed point solution of an equation system. The solution may, in general, be uncomputable. Thus, in Section 5.9, we present an abstract module semantics which over-approximates the concrete module semantics and allows to *compute* conservative module invariants.

Memory states. We define the module semantics over *trimmed memory states*. (The domain of trimmed memory states was already defined in Section 5.7.2.1). Thus, we refer to the concrete module semantics as the *trimming semantics* ($\mathcal{TSS}^\star$).

Operational Semantics. We do not explicitly define the operational semantics over trimmed memory states. Instead, we reuse the definition of $\mathcal{DOS}$'s operational semantics by *fabricating* a $\mathcal{DOS}$ memory state for every trimmed memory state. We apply the $\mathcal{DOS}$ operational semantics to the fabricated $\mathcal{DOS}$ memory state and trim the resulting state to obtain the effect of the statement.

Outline. The rest of this section is organized as follows: In Section 5.8.1, we provide some preliminary definitions regarding similarity between components and memory states. In Section 5.8.2, we define the meaning of *intra*module operations (i.e., intraprocedural statements and *intra*module procedure calls.). In Section 5.8.3, we define the meaning of *inter*module procedure calls made *by* the module. This allows us to handle the second challenge posed in Section 5.3, i.e., how to handle invocations to other modules without reanalyzing them. In Section 5.8.4, we show how we determine all calling contexts to a module without knowing its clients. In particular, we provide a possible answer to the first challenge posed in Section 5.3, i.e., how to determine all calling contexts to a module without knowing its clients. In Section 5.8.5, we define the system of equations whose least fixpoint is the module semantics.

5.8.1 Component Similarity and Trimmed State Similarity

Two components are similar if they are of the same module and one component can be produced from the other one by consistently renaming the locations of the other component.

Definition 5.8.1 (Location renaming function) *A **location renaming function** is a bijective total function $i : Loc \cup \{null, \ominus\} \to Loc \cup \{null, \ominus\}$ which maps null to null and $\ominus$ to $\ominus$, i.e., $i(null) = null$ and $i(\ominus) = \ominus$.*

Definition 5.8.2 (Component Similarity) *Component c_1 and $c_2 = \langle I_2, L_2, R_2, h_2, t_2, m_2 \rangle$ are **similar according to a renaming function** i, denoted by $c_1 \overset{c}{\sim}_i c_2$, if*

- $I_1 = \{i(l) \mid l \in I_2\}$,
- $L_1 = \{i(l) \mid l \in L_2\}$,
- $R_1 = \{i(l) \mid l \in R_2\}$,
- $h_1 = i \circ h_2 \circ i^{-1}$,
- $t_1 = t_2 \circ i^{-1}$, *and*
- $m_1 = m_2$.

*Components c_1 and c_2 are **similar**, denoted by $c_1 \overset{c}{\sim} c_2$, if there exists a renaming function i such that $c_1 \overset{c}{\sim}_i c_2$.*

Two trimmed states are said to be similar if both the component *and* the environment of one trimmed memory state can be produced from the other by the same location renaming function.

Definition 5.8.3 (Trimmed-states Similarity) *Trimmed-states $\sigma_1^\star = \langle \rho_1, c_1^\star \rangle$ and $\sigma_2^\star = \langle \rho_2, c_2^\star \rangle$ are **similar according to a renaming function** i, denoted by $\sigma_1^\star \overset{t}{\sim}_i \sigma_2^\star$, if $\rho_1 = i \circ \rho_2$ and $c_1^\star \overset{c}{\sim}_i c_2^\star$. Trimmed-states $\sigma_1^\star$ and $\sigma_2^\star$ are **similar**, denoted by $\sigma_1^\star \overset{t}{\sim} \sigma_2^\star$, if there exists a renaming function i such that $\sigma_1^\star \overset{t}{\sim}_i \sigma_2^\star$.*

We say that two $\mathcal{DOS}$ memory states $\sigma_1, \sigma_2 \in \Sigma_D$ are *similar under the trimming abstraction*, denoted by $\sigma_1 \overset{t}{\sim} \sigma_2$, if $trim(\sigma_1) \overset{t}{\sim} trim(\sigma_2)$. We say that two set of $\mathcal{DOS}$ memory states $\mathrm{S}_1, \mathrm{S}_2 \subseteq \Sigma_D$ are *similar under the trimming abstraction*, denoted by $\mathrm{S}_1 \overset{t}{\sim} \mathrm{S}_2$, if for every state $\sigma_1 \in \mathrm{S}_1$ there exists a state $\sigma_2 \in \mathrm{S}_2$ such that $\sigma_1 \overset{t}{\sim} \sigma_2$, and vice versa.

5.8.2 Meaning of Intramodule Statements

In this section, we define the meaning of intramodule program statements in the trimming semantics. The semantics is defined using a transition relation $\overset{\star}{\rightsquigarrow} \subseteq \Sigma^\star \times Stmt \times \Sigma^\star$. We define the meaning of intramodule statements by *constructing* for every trimmed state $\sigma^\star \in \Sigma^\star$ a $\mathcal{DOS}$ memory state $\sigma \in \Sigma_D$ for which $\sigma^\star \overset{t}{\sim} trim(\sigma)$, which we refer to as the *minimal $\mathcal{DOS}$ state abstracted by $\sigma^\star$*, and applying the $\mathcal{DOS}$ operational semantics to this state. Interestingly, we can define the effect of an intra-module statement st in the trimming semantics on an arbitrary trimmed state $\sigma^\star \in \Sigma^\star$ by applying st to *any* of the $\mathcal{DOS}$ memory states $\sigma \in \Sigma_D$ for which $\sigma^\star \overset{t}{\sim} trim(\sigma)$. Intuitively, we can do so because intramodule statements are insensitive to the contents of the heap *outside* the current component.

5.8.2.1 Minimal $\mathcal{DOS}$ States

We construct the minimal $\mathcal{DOS}$ memory state which abstracted by a given trimmed memory state $\sigma^\star = \langle \rho, c^\star \rangle$ by, basically, constructing a $\mathcal{DOS}$ memory state σ in which all the locations inside $c^\star$ are in σ's current component and every location in the rim of $c^\star$ is placed in its own sealed component.

Definition 5.8.4 (Minimal $\mathcal{DOS}$ states) *The **minimal $\mathcal{DOS}$ memory state corresponding to a trimmed memory state** $\sigma^\star = \langle \rho, \langle I, L, R, h, t, m \rangle \rangle \in \Sigma^\star$, denoted by $dos(\sigma^\star)$, is $dos(\sigma^\star) = \langle \rho, L \cup R, h, t, m \rangle$.*

Remark 5.8.5 *Note that a minimal $\mathcal{DOS}$ state may never arise in* any *program. However, it is a dynamically encapsulated $\mathcal{DOS}$ memory state which satisfies all the requirements of Invariant 1.*

5.8.2.2 The Meaning of Intraprocedural Statements

The effect of an intraprocedural statement st on a trimmed state $\sigma^\star \in \Sigma^\star$ is defined by the effect of st according to the $\mathcal{DOS}$ semantics on $dos(\sigma^\star)$, the minimal $\mathcal{DOS}$ memory state pertaining to $\sigma^\star$, i.e.,

$$\langle \sigma^\star, st \rangle \overset{\star}{\rightsquigarrow} trim(\sigma') \iff \langle dos(\sigma^\star), st \rangle \overset{DOS}{\rightsquigarrow} \sigma'$$

The following lemma shows that the meaning of intraprocedural statements in the trimming semantics provides the same meaning (up to similarity) for every two $\mathcal{DOS}$ memory state which are similar under the trimming abstraction. The lemma follows immediately from a case analysis of the definition of the different intraprocedural statements in the $\mathcal{DOS}$ semantics (see Section 5.5) and the trimming abstraction (see Section 5.7.2.1).

Lemma 5.8.6 *Let $st \in Stmt$ be an intraprocedural program statement. For any $\sigma_1, \sigma_2 \in \Sigma_D$ such that $\sigma_1 \overset{t}{\sim} \sigma_2$ it holds that*

$$\{\sigma_1' \mid \langle \sigma_1, st\rangle \overset{DOS}{\rightsquigarrow} \sigma_1'\} \overset{t}{\sim} \{\sigma_2' \mid \langle \sigma_2, st\rangle \overset{DOS}{\rightsquigarrow} \sigma_2'\} .$$

5.8.2.3 The Meaning of Intramodule Procedure Invocation: Call

We define the meaning of an intramodule procedure of an arbitrary procedure invocation $y = p(x_1, \ldots, x_k)$ according to the trimming semantics using a transition relation $\overset{\star}{\rightsquigarrow} \subseteq \Sigma^\star \times Call_{y=p(x_1,\ldots,x_k)} \times \Sigma^\star$.

The effect of a call statement $Call_{y=p(x_1,\ldots,x_k)}$ on a trimmed (call) state $\sigma^{\star c} \in \Sigma^\star$ is defined by the effect of st according to the $\mathcal{DOS}$ semantics on $dos(\sigma^{\star c})$, the minimal $\mathcal{DOS}$ memory state pertaining to $\sigma^{\star c}$, i.e.,

$$\langle \sigma^{\star c}, Call_{y=p(x_1,\ldots,x_k)}\rangle \overset{\star}{\rightsquigarrow} trim(\sigma^e) \iff \langle dos(\sigma^{\star c}), Call_{y=p(x_1,\ldots,x_k)}\rangle \overset{DOS}{\rightsquigarrow} \sigma^e$$

The following lemma shows that the meaning of a $Call$ operation pertaining to an intramodule procedure call in the trimming semantics provides the same meaning (up to similarity) for every two $\mathcal{DOS}$ memory states which are similar under the trimming abstraction. The lemma follows immediately from the definition of the $\mathcal{DOS}$ semantics (see Section 5.5) and the trimming abstraction (see Section 5.7.2.1).

Lemma 5.8.7 *Let $y = p(x_1, \ldots, x_k)$ be an arbitrary procedure invocation. For any $\sigma_1^c, \sigma_2^c \in \Sigma_D$ such that $\sigma_1^c \overset{t}{\sim} \sigma_2^c$ it holds that*

$$\{\sigma_1^e \mid \langle \sigma_1^c, Call_{y=p(x_1,\ldots,x_k)}\rangle \overset{DOS}{\rightsquigarrow} \sigma_1^e\} \overset{t}{\sim} \{\sigma_2^e \mid \langle \sigma_2^c, Call_{y=p(x_1,\ldots,x_k)}\rangle \overset{DOS}{\rightsquigarrow} \sigma_2^e\} .$$

5.8.2.4 The Meaning of Intramodule Procedure Invocation: Ret

We define the meaning of a return statement from an intramodule procedure invocation of an arbitrary procedure invocation $y = p(x_1, \ldots, x_k)$ according to the trimming semantics using a transition relation $\overset{\star}{\rightsquigarrow} \subseteq \Sigma^\star \times \Sigma^\star \times Ret_{y=p(x_1,\ldots,x_k)} \times \Sigma^\star$.

The effect of a return statement $Ret_{y=p(x_1,\ldots,x_k)}$ on a trimmed call state $\sigma^{\star c} \in \Sigma^\star$ and a trimmed exit state $\sigma^{\star x} \in \Sigma^\star$ is defined by the effect of st according to the $\mathcal{DOS}$ semantics on $dos(\sigma^\star)$, the minimal $\mathcal{DOS}$ memory state pertaining to $\sigma^\star$. However, here we have to be careful not to introduce clashes between location names in the call- and exit- states. Specifically, we need to make sure that memory locations which are used by the caller, but were irrelevant for the call were not allocated by the callee. (See also Section 5.6.3). Thus, we define the effect of a return statement $Ret_{y=p(x_1,\ldots,x_k)}$ on a trimmed call state $\sigma^{\star c} \in \Sigma^\star$ and a trimmed exit state $\sigma^{\star x} \in \Sigma^\star$ by applying the meaning of the Ret in the $\mathcal{DOS}$ semantics to (any) minimal $\mathcal{DOS}$ states which are (i) possible call- and exit- states to the memory states for the invocation (see Definition 5.6.9) and (ii) pertain to trimmed memory states which are similar to $\sigma^{\star c}$ and $\sigma^{\star x}$.

$$\langle \sigma^{\star c}, \sigma^{\star x}, Ret_{y=p(x_1,\ldots,x_k)}\rangle \overset{\star}{\rightsquigarrow} trim(\sigma^r) \iff \langle dos(\widetilde{\sigma^{\star c}}), dos(\widetilde{\sigma^{\star r}}), Ret_{y=p(x_1,\ldots,x_k)}\rangle \overset{DOS}{\rightsquigarrow} \sigma^r$$

where

$\widetilde{\sigma^{\star c}} \overset{t}{\sim} \sigma^{\star c}$,

$\widetilde{\sigma^{\star x}} \overset{t}{\sim} \sigma^{\star x}$,

$dos(\widetilde{\sigma^{\star c}})$ and $dos(\widetilde{\sigma^{\star r}})$ are possible call- and exit- states for the intramodule invocation $y = p(x_1, \ldots, x_k)$.

The following lemma shows that the meaning of a Ret operation pertaining to an intramodule procedure call in the trimming semantics provides the same meaning (up to similarity) for every two $\mathcal{DOS}$ memory states which are similar under the trimming abstraction (and possible call- and return- states for the invocation). The lemma follows immediately from the definition of the $\mathcal{DOS}$ semantics (see Section 5.5) and the trimming abstraction (see Section 5.7.2.1) using the same observations as in Lemma 5.6.10.

Lemma 5.8.8 *Let $y = p(x_1, \ldots, x_k) \in Stmt$ be an arbitrary intra-module procedure invocation. For any $\sigma_1^c, \sigma_1^x, \sigma_2^c, \sigma_2^x \in \Sigma_D$ such that: (i) σ_i^c and σ_i^x are possible call- and exit- memory states for st (for $i = 1, 2$), (ii) $\sigma_1^c \overset{t}{\sim} \sigma_2^c$, and (iii) $\sigma_1^x \overset{t}{\sim} \sigma_2^x$. it holds that*

$$\{\sigma_1^r \mid \langle \sigma_1^c, \sigma_1^x, Ret_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} \sigma_1^r\} \overset{t}{\sim} \{\sigma_2^r \mid \langle \sigma_2^c, \sigma_2^x, Ret_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} \sigma_2^r\} .$$

5.8.3 The meaning of Intermodule Procedure Calls Made by the Module

In this section, we define the meaning of intermodule procedure calls made *by* the module. Recall that we assume that the module dependency graph is acyclic. Thus, we can assume that the semantics of the module of the invoked procedure is already computed.

At this point, we might be tempted to use the internal invariants of the invoked module which summarizes its procedure for every possible calling context that might arise in a dynamically encapsulated execution. However, this approach has one caveat: The implementation invariants of a module are expressed using trimmed memory states of that module. Thus, using them directly would undermine our goal to define the module semantics in such a way that it is not aware of the contents of the parts of the heap manipulated by other modules. (Informally, this situation arises because in the $\mathcal{DOS}$ semantics, executing the Ret operation pertaining to an intermodule procedure invocation requires considering information about the contents of heap parts manipulated by *different* modules. Specifically, the Ret operation is aware of the *contents of the current components* of both the caller and of the callee).

Instead of using the internal module invariants of the invoked module, we compute the effect of an intermodule call in the trimming semantics using a "shallow" summary of the invoked procedure. This summary records only the effect of the procedure invocation on the caller's without exposing the internal structure of the components of the invoked module. More specifically, we exploit the limited effect of intermodule procedure invocations on the caller's current component: The only effect an intermodule procedure call has on the current component of the caller is that (i) dangling references are blocked and (ii) the return value is assigned to a local variable.[15] Thus, we extract out of the internal module invariants of the callee a "shallow" summary which suffice to capture these effects.

5.8.3.1 Shallow Procedure Summaries

In this section, we define a semantic notion of shallow memory states and shallow procedure summaries.

We construct the shallow trimmed memory state corresponding to a trimmed memory state by, essentially, restricting the trimmed memory state to contain only objects which are pointed to by variables.

Definition 5.8.9 (Shallow Trimmed Memory States) *A **shallow trimmed memory state** is a trimmed memory state in which every object is pointed to by a variable and the heap is undefined. The **shallow trimmed memory state corresponding to a trimmed memory state** $\sigma^\star = \langle \rho, \langle I, L, R, h, t, m \rangle \rangle$, denoted by $shallow(\sigma^\star)$ is $\langle \rho, \langle I \cap E, L \cap E, R \cap E, \bot, t|_E, m \rangle \rangle$ where $E = \{\rho(x) \in Loc \mid x \in dom(\rho)\}$.*

Definition 5.8.10 (Shallow procedure specification) *Given a procedure p of a module m the **Shallow procedure summary** for a procedure p, denoted by $[\![p]\!]^s_\star$, is*

$$[\![p]\!]^s_\star = \left\{ \langle \sigma^\star, \sigma^{\star\prime} \rangle \;\middle|\; \begin{array}{c} \langle \sigma, \sigma' \rangle \in [\![Inv_m^{imp}\ p]\!], \\ \sigma^\star = shallow(trim(\sigma)), \\ \sigma^{\star\prime} = shallow(trim(\sigma')) \end{array} \right\} .$$

[15] By our simplifying assumptions, the return value must point either to a parameter object or to a component not previously owned by the caller. The latter case amounts to a new object in the rim of the caller's current component.

Any set $S \subseteq \Sigma^\star \times \Sigma^\star$ which is a superset of $[\![p]\!]^s_\star$ is a ***sound trimmed shallow specification of a procedure*** p.

The shallow procedure specification records all possible pre and post shallow trimmed states that may occur in any dynamically encapsulated program.

We assume that we can construct arbitrary shallow memory states. Thus, given the specification, as defined in Section 5.4.2, of an arbitrary procedure p, we can construct a sound shallow specification of procedure p.

5.8.3.2 Using the Shallow Procedure Summaries

We compute the effect of an intermodule procedure invocation on the caller's trimmed memory state using the shallow procedure specification of the callee. Specifically, the effect of an intermodule procedure invocation $y = p(x_1, \ldots, x_k)$ on a trimmed (call) state $\sigma^{\star c} \in \Sigma^\star$ is defined by computing the entry state according to the $\mathcal{DOS}$ semantics which results when the call is invoked on $dos(\sigma^{\star c})$, the minimal $\mathcal{DOS}$ memory state pertaining to $\sigma^{\star c}$, and using the shallow procedure summary to determine the resulting exit state. Note that by definition of a minimal $\mathcal{DOS}$ memory state, trimming the computed entry state results in a shallow trimmed memory state. Thus, we use the shallow procedure summary of the invoked procedure to determine the exit state.

$$\langle \sigma^{\star c}, Call_{y=p(x_1,\ldots,x_k)} \rangle \overset{\star}{\rightsquigarrow} trim(\sigma^r) \iff \begin{array}{c} \langle dos(\sigma^{\star c}), Call_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} \sigma^e \\ \langle trim(\sigma^e), \sigma^x_s \rangle \in [\![p]\!]^s_\star \\ \langle dos(\sigma^{\star c}), dos(\sigma^x_s), Ret_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} \sigma^r \end{array}$$

The following lemma formalizes the soundness of using the procedure summaries. It follows immediately from the definition of the $\mathcal{DOS}$ semantics and the above observations.

Lemma 5.8.11 *Let p be a procedure of module m. Let P be a program using m. Let σ_c, σ_e, σ_x, and σ_r be $\mathcal{DOS}$ memory states that arise during an intermodule invocation of p. The following holds:*

(1) $\langle dos(trim(\sigma_c)), Call_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} \sigma^e_s$ *and* $trim(\sigma^e_s) = shallow(trim(\sigma^e))$.

(2) $\langle shallow(trim(\sigma^e)), shallow(trim(\sigma^x)) \rangle \in [\![p]\!]^s_\star$.

(3) $\langle dos(trim(\sigma_c)), dos(shallow(trim(\sigma^x))), Ret_{y=p(x_1,\ldots,x_k)} \rangle \overset{DOS}{\rightsquigarrow} dos(trim(\sigma^r))$

Lemma 5.8.11 states that: (1) applying the $Call$ rule for the trimmed call memory state results in the shallow memory state pertaining to the entry state; (2) the shallow states pertaining to the pair of entry and exit state is in the shallow procedure summary; and (3) combining the trimmed call memory state with the exit state results in the trimmed return memory state.

5.8.4 The Meaning of Intermodule Procedure Calls to the Module

In this section, we show how to determine all possible calling contexts (i.e., input memory states) that can arise in an intermodule procedure call to a procedure of the module in any dynamically encapsulated program. Thus, we provide a possible answer to the first challenge posed in Section 5.3.

The main idea is to utilize the following properties of the $\mathcal{DOS}$ semantics:

- In an intermodule procedure invocation, the actual parameters can only point to sealed components. Thus, the current component of an entry memory state in an intermodule procedure call is comprised of a disjoint union of *already generated sealed components* of the module.
- The only way a sealed component of module m can be created is when a procedure of a module returns. Similarly, such a component can be mutated only when it is being passed as a parameter to a procedure of of module m.

Informally, the above properties allow us to *anticipate the possible entry memory states of an intermodule procedure call* by collecting all possible sealed components that are generated at the exit-sites of intermodule procedure calls and use them to *fabricate* the input states to the intermodule procedure calls.

5.8.4.1 Fabricating Memory States

We now define certain predicates and operations which operate on components and trimmed memory states. These operations allow us to explicitly fabricate memory state.

Definition 5.8.12 (Disjointness of components) *Components c_1 and c_2 are **disjoint**, denoted by $c_1 \sharp c_2$, if $(L_1 \cup R_1) \cap (L_2 \cup R_2) = \emptyset$.*

Definition 5.8.13 (Combination of components) *The **combination** of disjoint components c_1 and c_2 denoted by $c_1 \oplus c_2$, is the component $\langle I_1 \cup I_2, L_1 \cup L_2, R_1 \cup R_2, h_1 \cup h_2, t_1 \cup t_2, m\rangle$.*

Definition 5.8.14 (Empty component) *The **empty component of module** m, denoted by $c^m_\emptyset$, is $c^m_\emptyset = \langle \emptyset, \emptyset, \emptyset, \bot, \bot, m\rangle$,*

Definition 5.8.15 (Rim component) *The **rim component** of a location l, a type T, and module m, such that $m(T) \neq m$, denoted by $c^m_{rim}(l, T)$, is $c^m_{rim}(l, T) = \langle \{l\}, \emptyset, \{l\}, \bot, [l \mapsto T], m\rangle$.*

We say that a component $c \in \mathcal{C}(\sigma)$ is the *component of a (reachable) location $l \in \mathcal{R}(\sigma)$ in memory state σ*, denoted by $c_\sigma(l)$, if l is inside c.

Definition 5.8.16 ($\mathcal{M}$-projected component) *The **$\mathcal{M}$-projected component** of a variable x in a $\mathcal{DOS}$ memory state $\sigma = \langle \rho, L, h, t, m\rangle$ with respect to module m' such that $m(t(x)) \neq m$, denoted by $c^{m'}_\sigma(x)$, is*

$$c^{m'}_\sigma(x) = \begin{cases} c_\sigma(\rho(x)) & m(t(x)) = m' \text{ and } \rho(x) \in Loc \\ c^{m'}_{rim}(\rho(x), t(x)) & m(t(x)) \neq m' \text{ and } \rho(x) \in Loc \\ c^{m'}_\emptyset & \text{otherwise .} \end{cases}$$

The $\mathcal{M}$-projected component of a variable x in memory state is:
- the sealed component whose header, $\rho(x)$, is pointed to by x, if x points to a location of module $m' \neq m$;
- the rim component of the location pointed to by x, $\rho(x)$, and x's type, $t(\rho(x))$, if $t(\rho(x)) \neq m'$; or
- the empty component of module m', otherwise (i.e., if x does not point to a location).

Definition 5.8.17 (Projection of Components) *The **projection** of component $c = \langle I, L, R, h, t, m\rangle$ on an entry-location $l \in I$, denoted by $c|_l$, is the component $\langle I \cap L_{rel}, L \cap L_{rel}, R \cap L_{rel}, h|_{L_{rel}}, t|_{L_{rel}}, m\rangle$, where $L_{rel} = R_h(\{l\})$.*

5.8.4.2 Fabricating Input Memory States

Lemma 5.8.18 formalizes the intuition that we discussed above, i.e., that the current component of every entry state to an intermodule procedure call is comprised of some sealed components at the call state.

Lemma 5.8.18 (Inter-module entry state fabrication) *Let $\sigma_c \in \Sigma_D$ be a possible call memory state for a procedure invocation $y = p(x_1, \ldots, x_k)$ such that $m(\sigma_c) \neq m(p)$. Let $c_i = c^{m(p)}_{\sigma_c}(\rho_c(x_i))$, for $i = 1, \ldots, k$. Let σ_e be the entry state resulting from the invocation $y = p(x_1, \ldots, x_k)$ on σ_c, i.e., $\langle \sigma_c, Call_{y=p(x_1,\ldots,x_k)}\rangle \overset{DOS}{\rightsquigarrow} \sigma_e$. Let $\sigma^\star_e = \langle [z_i \mapsto \rho_c(x_i) \mid 1 \leq i \leq k], c_1 \oplus \ \ldots \oplus c_k\rangle$ be a fabricated memory state. Then, $\sigma^\star_e = trim(\sigma_e)$.*

Sketch of Proof: The proof is based on the following properties of the $\mathcal{DOS}$ semantics: every formal parameter in an entry memory state σ_e resulting from an inter-module procedure call dominates its reachable subheap. In addition, in σ_e, different formal parameters point to disjoint subheaps. Also procedures of module m can manipulate only memory states whose current component is of module m.

Thus, in any trimmed memory state $\sigma^\star_e$ such that $trim(\sigma_e) \overset{t}{\sim} \sigma^\star_e$, a formal parameter whose type is not of module m, points to an isolated object whose fields are undefined. The current component is the only component of module m. It is implicitly created by reassigning to the current component of the entry memory state all the locations inside the subcomponents of the current component of the call memory state whose headers are by an actual parameter. In particular, the subheap comprising every such subcomponent is not mutated.

We utilize Lemma 5.8.18 to conservatively determine every possible input state to a procedure. We assume that we can fabricate empty components and rim components of arbitrary modules. The challenge is to determine the possible sealed components of (the analyzed) module m. Lemma 5.8.19 shows that in any program, such components were sealed when a preceding inter-module procedure call was invoked. Furthermore, no other module could have modified these components, as guaranteed by the program model (see Section 5.4).

Lemma 5.8.19 (Consistency of sealed components) *Let $\pi \in \Pi^P_{\mathcal{DOS}}$ be a feasible execution of an arbitrary program P according to the $\mathcal{DOS}$ semantics. Let $c \in \mathcal{C}(\sigma)$ be a sealed component of module m in the $\mathcal{DOS}$ memory state $\sigma = out(\pi)$. Then, there exists a prefix π' of π such that*

(i) $\langle e_p, \sigma_x \rangle = top(\pi(|\pi'|))$ is an exit program state of some procedure p of module m.

(ii) $m(cur_{proc}(pop(\pi(|\pi'|)))) \neq m$, the caller of p is not of module m.

(iii) There exists a location l which in memory state σ_x is pointed to by a non-transferred formal variable or by the ret variable (the return value), $l \in \{\rho_x(ret), \rho_x(z_i) \mid 1 \leq i \leq k\} \cap Loc$ (assuming that p's formal variables are $z_1, \ldots, z_k$), such that $c_\sigma(l) \sim c^\star(\sigma_x)|_l$.

5.8.5 A Least Fixpoint Definition of the Module Semantics

In this section, we present an equation system whose (fixpoint) solution defines the program independent meaning of an arbitrary module m. The equation system is specified based on the trimming semantics and the *bodies* of the procedures of module m (Section 5.4). The effect of intermodule procedure calls made by procedures of module m is determined using the (pre-computed) shallow summaries of the invoked procedures (Section 5.4.2).

The equation system is, in most parts, a standard data-flow-like based equation system, aimed at determining the collecting semantics (relational trimming-semantics, in our case) of a program. The challenge is that we do not want to analyze the module in the context of a *specific* program but in the context of *any* program, nor do we want to analyze the procedures of modules used by m. Section 5.8.4 describes how we achieve the above by *fabricating* every possible entry memory state to any procedure of module m in any program. Section 5.8.3.1 describes how we utilize pre-computed shallow procedure summaries to handle intermodule procedure calls.

Figure 5.6 provides an equation system whose least fixpoint solution determines the module semantics. For simplicity, and without loss of generality, we assume that the procedures of module m are either *interface procedures*, which may be invoked by procedures from modules other than m, or *private procedures* which may be invoked only by procedures from module m.

The equation system utilizes trimmed shallow procedure summaries to handle inter-module procedure calls. It determines all possible entry memory states for every procedure of module m by combining, in every possible way, the sealed components that were already found. Lemma 5.8.11 ensures soundness of the utilization of the procedure specification. Lemma 5.8.18 and Lemma 5.8.19 ensure that all possible input states are found.

Note that the ownership transfer specification of procedures of module m (Section 5.4.2) can be *conservatively verified* using any given fixpoint solution to the equation system shown in Figure 5.6.

We note that although the equation system is defined in terms of trimmed memory states of (*only*) module m. It does not lead to an *effective* modular analysis *algorithm*: In general, trimmed memory states might be of an unbounded size. Thus, an algorithmic solution might be out of hand. However, one can be derived, if the trimming semantics is replaced by a semantics which approximates it in a bounded way, e.g., see Section 5.9.

5.9 Abstract Module Semantics

This section presents an approach for a static analysis which conservatively identifies *conditional module invariants* and verifies that the procedure of a module resects their ownership transfer specification in *any dynamically encapsulated* programs.

The analysis is derived by two (successive) abstractions of the $\mathcal{DOS}$ semantics: The *trimming semantics* provides the basis of our *modular* analysis by explicitly representing only components of the analyzed module. The *abstract trimming semantics* allows for an effective analysis by providing a *bounded* abstraction of trimmed memory states (utilizing existing *intra*procedural abstractions).

Intraprocedural

$$[\![n']\!]_\star = \bigcup_{\langle n,n'\rangle \in E_p} [\![n]\!]_\star \circ [\![stmt_{G_p}(\langle n, n'\rangle)]\!]_\star \qquad n' \neq s_p, n' \neq e_p, n' \text{ is not a return-site}$$

Interprocedural Intra-module (invocation of private procedures)

$$[\![s_p]\!]'_\star = \bigcup_{\langle n,n'\rangle \in E_q} \{\langle \sigma^{\star\prime\prime}, \sigma^{\star\prime\prime\prime}\rangle \mid \langle \sigma^\star, \sigma^{\star\prime\prime}\rangle \in [\![n]\!]_\star \circ [\![Call_{y=p(x_1,\ldots,x_k)}]\!]_\star\} \qquad m(q) = m(p) = m,\ stmt_{G_p}(\langle n, n'\rangle) = invoke$$

$$[\![n']\!]_\star = [\![n]\!]_\star \circ \left\{ \langle \sigma^\star, \sigma^{\star\prime}\rangle \;\middle|\; \begin{array}{c} \sigma^{\star\prime} \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma^\star, \sigma^{\star\prime\prime}), \\ \sigma^{\star\prime\prime} \in [\![e_p]\!]_\star([\![Call_{y=p(x_1,\ldots,x_k)}]\!](\sigma^\star)) \end{array} \right\} \quad \begin{array}{l} \langle n, n'\rangle \in E_q, m(p) = m(q) = m \\ stmt_{G_p}(\langle n, n'\rangle) = invoke \end{array}$$

Interprocedural Inter-module (calls to lower modules)

$$[\![n']\!]_\star = [\![n]\!]_\star \circ \left\{ \langle \sigma^\star, \sigma^{\star\prime}\rangle \;\middle|\; \begin{array}{c} \sigma^{\star\prime} \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma^\star, \sigma^{\star\prime\prime}), \\ \sigma^{\star\prime\prime} \in [\![p]\!]^s_\star([\![Call_{y=p(x_1,\ldots,x_k)}]\!](\sigma^\star)) \end{array} \right\} \quad \begin{array}{l} \langle n, n'\rangle \in E_q, m(p) \neq m \\ stmt_{G_p}(\langle n, n'\rangle) = invoke \end{array}$$

Interprocedural Inter-module (simulating external calls to interface procedures)

$$[\![s_p]\!]_\star = \left\{ \langle \sigma^\star, \sigma^\star\rangle \;\middle|\; \begin{array}{c} c_i \in [\![t_p(z_i)]\!]_\star \text{ for } i = 1, \ldots, k, \\ \sigma^\star = \langle [z_i \mapsto hdr(c_i) \mid 1 \leq i \leq k], c_1 \oplus \ \ldots \oplus c_k\rangle \end{array} \right\} \qquad m(p) = m$$

Sealed microheaps

$$[\![T]\!]_\star = \bigcup_{m(q)=m} \left\{ c'|_l \;\middle|\; \begin{array}{c} \langle \sigma^\star, \sigma^{\star\prime}\rangle \in [\![e_p]\!]_\star,\ \sigma^{\star\prime} = \langle \rho', c'\rangle,\ t'(l) = T, \\ l \in \{\rho'(ret), \rho'(x) \mid x \in F_q\} \cap Loc \end{array} \right\} \qquad m(T) = m$$

Figure 5.6: An equation system whose (fixpoint) solution determines sound module invariants for module m. We assume p is a procedure with k formal parameters, $z_1, \ldots, z_k$. $invoke \equiv y = p(x_1, \ldots, x_k)$. $c = \langle I, L, R, h, t, m\rangle$. We assume that $[\![T]\!]_\star = \{c^m_{rim}(l, T) \mid l \in Loc\}$ for all types T such that m uses $m(T)$. We denote the type of a local variable x of a procedure p by $t_p(x)$.

5.9.1 Abstract Trimming Semantics

We provide an effective conservative abstract interpretation [CC77] algorithm which determines module invariants by devising a bounded abstraction of trimmed memory states. Rather than providing a new intraprocedural abstraction and analyses, we show how to *lift* existing *intra*procedural shape analyses to obtain a modular shape abstraction.

An abstraction of a trimmed memory state, being comprised of an environment of a single procedure and a subheap, is very similar to an abstraction of a standard two-level store. The additional elements that the abstraction needs to track is a bounded number of entry-locations and a distinction between internal objects and rim objects. In addition, the abstract domain, expected to support operations pertaining to basic pointer manipulating statements, should also allow for:

1. the operations required for cutpoint-free local-heap analysis: carving out subheaps reachable from variables and combining disjoint subheaps;
2. the ability to answer queries regarding domination by variables;
3. checking if a $\ominus$-valued reference is accessed; and
4. setting the values of all reference pointing to a given variable-pointed object to $\ominus$ (i.e., the *blocking* operation).

The abstract domains of [LAIS06, MYRS05, DOY06], which already support the operations required for performing standard cutpoint-free local-heap analysis (see Section 4.10.1) handle the first two operations and can be easily extended to support the other two.

Example: Abstract Trimming Semantics via Canonical Abstraction

In this section, we sketch, as an example, how to define a *bounded* parametric abstraction for trimmed memory states using canonical abstraction.

We abstract sets of trimmed memory states by a point-wise application of an *extraction function* $\beta^\star \colon \Sigma^\star \to \mathit{3Struct}$ (e.g., see [NNH99]) mapping a trimmed memory state $\sigma^\star$ to its *best* representation by an *abstract trimmed memory state* $\sigma^{\star\sharp}$. An abstract state $\sigma^{\star\sharp}$ provides a conservative bounded representation for the unbounded number of locations in every trimmed-state. We use set-union as the join-operator. Technically, abstract trimmed memory states are represented using 3-valued logical structures. The tracked properties are encoded as predicates.

We define a Galois connection between the powerset domain of trimmed memory states and $\mathit{3Struct}$ using a *representation function* (see Section 2.5.1) $\beta^\star \colon \Sigma^\star \to \mathit{3Struct}$ which maps a program state to its "most-precise representation" in $\mathit{3Struct}$. Function $\beta^\star$ is a composition of two functions:

1. *trimmedTo2VLS*$\colon \Sigma^\star \to \mathit{2Struct}$ which maps a trimmed memory state to an unbounded 2-**valued logical structure** S, representing it.
2. *canonical abstraction* which conservatively bounds the resulting 2-valued logical structure by a 3-valued logical structure.

The Galois connection between the power-domain of trimmed memory states and the power-domain of 3-valued logical structures $(2^{\Sigma_D}, \alpha \colon 2^{\mathit{3Struct}} \to 2^{\mathit{3Struct}}, \gamma \colon 2^{\mathit{3Struct}} \to 2^{\mathit{3Struct}}, 2^{\mathit{3Struct}})$ is defined in a standard manner:

$$\alpha(\mathrm{S}) = \{\beta^\star(\sigma^\star) \mid \sigma^\star \in S^\star\} \text{ and } \gamma(SS) = \{\sigma^\star \in S^\star \mid S^\sharp \in SS,\ \beta^\star(\sigma^\star) \sqsubseteq S^\sharp\}\ .$$

The function *trimmedTo2VLS* is defined in a similar manner to the *to2VLS*^{cpf} function, defined in Figure 3.12. This function maps a trimmed memory state $\sigma^\star = \langle \rho, c^\star \rangle \in \Sigma_{L_{CPF}}$ to a *2-valued* logical structure S. Every object $o \in A$ is represented by a unique node in U^S. Tracked properties of the memory state are recorded by the predicates given in Figure 2.11, whose intended meaning is explained in Section 2.5.1.2. We also use the predicates inUc and inUx, shown in Figure 3.11 to implement the call rule as explained in Section 3.7.2.2.

For brevity, we do not repeat all the details of the definition of the set of properties or of the mapping of a trimmed memory state into a 3-value logical structure. We only note the additional predicates required to record the trimmed memory states. Specifically, these predicates, shown in Figure 5.7, record the type objects. (This information also allows us to distinguish between objects inside the components and objects at its rim and whether a reference parameter of a field is accessible or not.)

Abstracting sealed components. In addition, the abstraction needs to abstract (sets of) sealed components. To do so, we abstract sealed components in the same way that we abstract trimmed memory states, and referring to

Predicate	**Intended Meaning**
$T(v)$	v is an object of type `T`
$x_{\ominus}$	Reference variable `x` has an inaccessible value
$f_{\ominus}(v)$	Reference field `f` of object v has an inaccessible value

Figure 5.7: Additional core predicates used to represent a trimmed memory state using a 3-valued logical structure.

the header pointer, as essentially, a reference variable pointing into the component.

Chapter 6

Related Work

In this chapter, we review closely related work and contrast it with the results presented in this thesis.

6.1 Storeless Semantics and its Use in Program Analysis

In this section, we review existing storeless semantics and their use in program analysis.

6.1.1 Storeless Semantics

Storeless semantics was first introduced by Jonkers [Jon81]. The original motivation was to develop an *abstract model* [Ber79] for specifying the representation of abstract data types. The goal was to design an *abstract storage structure* which allows to specify representation of (unbounded) data structures in a level which is abstract enough to ignore low level concepts such as pointers and garbage collection, but at the same time general enough to allow description of arbitrary sharing and cyclicity. The main observation in [Jon81] is that a storage structure can be completely characterized by two things: the collection of its access paths and an equivalence relation which indicates whether two access paths lead to the same substructure. Jonkers has also envisioned that such structures can be used to specify the semantics of imperative programming languages with aliasing, sharing, and dynamic allocation. Indeed, this is the way in which we utilize his work.

Jonkers's storeless semantics does not handle procedure calls. A storeless semantics that handles procedure calls is defined by Deutsch in [Deu92a, Deu92b]. In Deutsch's semantics, the entire heap is explicitly represented in every state. Specifically, pending access paths are explicitly represented. In contrast, the storeless semantics presented in this thesis, $\mathcal{LSL}^{\text{CPF}}$ (see Section 3.4) and $\mathcal{LSL}$ (see Section 4.4), explicitly represent only a part of the heap (the procedure's local-heap). Specifically, neither $\mathcal{LSL}^{\text{CPF}}$ nor $\mathcal{LSL}$ explicitly represent pending access paths.

The development of a storeless semantics that represents only the procedure's local-heap (and, specifically, does not represent pending access paths) is challenging. In storeless semantics, allocated objects do not have unique immutable identifiers (e.g., an address). Instead, every dynamically allocated object O is represented by the set of pointer-access paths whose R-value equals O's L-value. In languages with destructive updates, a procedure can modify the R-value of access paths that start at variables of pending calls (i.e., pending access paths).

Deutsch's storeless semantics [Deu92a, Deu92b] handles the aforementioned problem by representing access paths that start from pending variables. Technically, the stack of activation records is represented as part of the state. (The stack is encoded, essentially, as a list). In contrast, our semantics only represents access paths that start from current variables. As a result, our semantics captures the procedure's local view of the memory, a view that includes only the procedure's local variables and only the objects that are reachable from these variables.[1]

Our main insight in the development of $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ is that the side-effects of a procedure invocation on R-values of pending access paths can be delayed to the procedure return—even though the memory cells do not

[1] Recall that we forbid global variables. The latter can be treated as value result parameters which are passed to every procedure.

have unique identifiers, e.g., locations.[2] The main idea is to track the effect of destructive updates on access paths that start with the set of objects that separate the part of the heap which the procedure can reach from the rest of the heap (objects that we call the *cutpoints* of the invocation). This allows our semantics to explicitly represent only current access paths (and to avoid explicitly representing pending access paths). A similar observation regarding the uniform effect of a procedure on pending access paths was made by [LR92, Deu94] for pointer analysis. We believe that our work is the first to explore the usage of this observation in semantics.

6.1.2 The Use of Storeless Semantics in Program Analysis

Venet [Ven99] presents an intraprocedural static pointer analysis for untyped languages which is based on storeless semantics. The analysis is capable of discovering potential sharing relationships among the data structures created by an imperative program. The analysis is able to distinguish between elements in inductively defined structures and does not require any explicit data type declaration by the programmer in order to construct the abstract interpretation. The price for the generality of the approach is in its precision. For example, it cannot capture the fact that a sorting algorithm does not create aliasing in the sorted list.

Intraprocedural storeless semantics is also used in [BIL03] to develop a logic that allows to express regular properties of unbounded data structures.

The interprocedural may-alias algorithm of [Deu94] uses a storeless representation of the heap. The algorithm is polynomial and can handle procedure calls, dynamic memory allocation and destructive updates. The algorithm is *not* shown to be an abstract interpretation of the storeless semantics defined in [Deu92a, Deu92b]. However, one can define a Galois connection between memory states in $\mathcal{LSL}$ with the abstract domain of [Deu94]; see Appendix D.4.

6.2 Shape Analysis

Shape analysis algorithms produce in compile time a description of the program's store at each program point. The description is conservative, i.e., it is valid for all possible executions of the program. For example, the analysis may reveal whether, at a certain program point, two or more pointers may point to the same memory object. In order to represent the structure of the program store, many shape analysis techniques use *shape-graphs*, e.g., [JM81, JM82, LH88, CWZ90, AW93, PCK93, SRW98, LARSW00, Yah01, SRW02]. Each node of a *shape-graph* represents one or more memory locations and each edge stands for one or more pointers. All the statements of the program are augmented with sets of *shape-graphs*. For each statement, every possible "shape" of the program's memory when the control reaches that statement is modeled by one of the *shape-graphs* in the set of *shape-graphs* which is associated with it. Thus, the entire set of *shape-graphs* associated with a statement constitutes a conservative approximation of all the possible "shapes" that the program memory can take at that point. (See [RSW04] for an introduction to shape analysis).

Parametric Shape Analysis

Shape analysis concerns the problem of determining shape invariants for programs that perform destructive updating on dynamically allocated storage. The complexity of the problem usually requires that algorithm be tuned for a specific class of data structures, e.g., lists or trees. (Specializing the analysis to handle specific data structure is especially useful when the analysis attempts to establish intricate reachability-based properties [LAIS06].) A parametric framework for shape analysis can be instantiated in different ways to create shape-analysis algorithms that provide different degrees of precision.

A parametric framework for shape analysis based on 3-valued logic was presented in [SRW02]. The framework can be instantiated in different ways by varying the tracked properties. We express our analyses in the framework of [SRW02], and realize them using its implementation in the TVLA system [LAS00].

[2] Having unique immutable addresses identifying objects greatly simplifies the task of designing a local-heap semantics. See, for example, the $\mathcal{LSB}$ semantics, defined in Section 2.3. Note, however, that an abstraction of an (unbounded) store-based heap is most likely to lose information pertaining to the (unbounded) number of addresses.

6.3 Interprocedural Shape Analysis

In this section, we review existing approaches for interprocedural shape analysis and contrast existing approaches for interprocedural shape analysis with ours. In Section 6.3.1, we provide a bird's-eye view of existing techniques for interprocedural program analysis which have been used for interprocedural *shape* analysis. In Sections 6.3.2, 6.3.3, we describe existing interprocedural *shape* analysis algorithms which are based on the *call string approach* and on the *functional approach* for interprocedural program analysis, respectively, and contrast them with our work. In Section 6.3.4, we discuss the (common aspects of the) shape abstractions which have been used by existing interprocedural, and in Section 6.3.5, we examine the notion of *heap modularity*, which is a common aspect of interprocedural shape analysis that are based on the functional approach.

6.3.1 Interprocedural Program Analysis

Static program analysis algorithms determine statically (i.e., without the programs being actually executed) dynamic properties of programs (i.e., properties of program executions). Interprocedural analysis concerns the static examination of programs consisting of multiple procedures. In contrast with intraprocedural dataflow analysis, where "precise" means "meet-over-all-paths" [Kil73], a precise interprocedural dataflow–analysis algorithm must provide the "meet-over-all-valid-paths" solution [SP81, RHS95].[3] Note that even a precise solution (i.e., the "meet-over-all-valid-paths" solution) may still include infeasible paths.

In the following, we review several standard techniques for interprocedural program analysis. Interprocedural shape analysis algorithms have been implemented using the *call string* [SP81] approach and the *functional approach* [SP81]. In our work, we use the functional approach.

6.3.1.1 The Call String Approach

The call string approach [SP81] approximates a small-step semantics. It separates call chains (and the related data flow information) that differ in a suffix of a fixed length. Thus, precision may increase with longer call strings. The maximally useful length for nonrecursive programs is the height of the call-graph. For recursive programs the number of possible call strings is infinite. Even if there are no recursive procedures, the number of call strings can be exponential in the number of procedures. Thus the call string length has to be limited [Mar99]. For infinite domains, in general, it is not possible to obtain a *precise* solution using this approach [SP81, Section 7.6]. However, for finite domains it is possible to obtain such a solution *even if the analyzed program is recursive* [SP81, Section 7.5].

6.3.1.2 The Functional Approach

The functional approach [CC78, SP81] approximates a large-step semantics (see, e.g., [Kah87, NNH99]). It computes a function for each procedure describing the "abstract" effect of the procedure. These functions are then used in a standard (intraprocedural-like) algorithm. Technically, this is achieved by lifting the dataflow problem from determining abstract values, i.e., dynamic properties, to finding a function that describes the accumulated abstract effect of all the valid paths from the entry of every procedure to each one of its statements. In particular, this approach requires an (efficient) composition of abstract functions [Mar99].

The functional approach allows computing a precise solution even when the abstract domain is infinite. Termination can be ensured by requiring that the lattice of *abstract functions* has a finite height. A common instance of the functional approach when the abstract domain is finite is to tabulate the procedure's input-output relation (see, e.g., [SP81, Section 7.3]).

6.3.1.3 Context Free Reachability

This approach is an efficient instantiation of the functional approach for certain types of problems, e.g., when using a powerset domain. In this approach, interprocedural analysis problems are converted into a special kind of graph-reachability problem: reachability along interprocedurally realizable paths. In contrast with ordinary

[3] A path is valid if it respects the fact that when a procedure finishes it returns to the site of the most recent call [SP81, Cal88, LR91, KS92, RHS95, SRH96].

reachability problems in directed graphs, realizable-path reachability problems involve some constraints in terms of which paths are considered. A realizable path mimics the call-return structure of a program execution, and only paths in which returns can be matched with corresponding calls are considered [RHS95, HRS95, SRH96].

Context free reachability was used, for example, in the *IFDS* framework, presented in [RHS95], for solving interprocedural problems where the abstract domain is finite and the transfer functions are distributive. The essence of their approach is the construction of an *exploded supergraph*. The exploded graph has $|D|$ nodes for every node in the program control flow graph (CFG), where D is the set of possible dataflow *facts*. Every edge e in the program (CFG) is replaced by a set of edges that explicitly represents the abstract effect of the statement annotating e (i.e., an edge e is replaced by the *graph* of the function of the statement annotating it). In this approach, an algorithm computes a precise solution to the dataflow problem by finding which nodes in the exploded supergraph are reachable from the nodes at the program start node. The nodes have to be reachable along paths that respect the context-free language of matching calls and returns (and hence the name "context-free reachability").

The iterative tabulation algorithm that we employ to implement our interprocedural shape analyses, as described in Sections 3.8 and 4.9, are based on the tabulation algorithm of the *IFDS* framework. We extended their tabulation algorithm to handle the case where procedures are passed only parts of the global state. (See Appendix F).

6.3.2 Interprocedural Shape Analysis using Call Strings

In our earlier work [RS01], we extended the 3-valued logic framework of [SRW02] to handle procedure calls in recursive programs manipulating singly linked lists using the call string approach. The main idea in [RS01] is to explicitly represent the runtime stack and to abstract it as a linked list. Despite the summarization of the values of local variables, the analysis is able to handle return statements from recursive calls rather precisely by tracking certain relationships between the stack and the heap pertaining to (all) pending variables.

We note that because the analysis of [RS01] represents the entire heap at every program point, the abstraction may lose information about properties of the heap *for parts of the heap that cannot be affected by the procedure at all*. In addition, the analysis of [RS01] requires the use of special predicates to record stack-heap relationship. In contrast, in our current work, we only record more "localized" relations pertaining to the *sharing patterns* between the irrelevant heap and the procedure's local-heap. Specifically, we never record relationships between pending variables and the irrelevant parts of the heap.

At the *concrete* level, our current analysis is indeed more local than [RS01], and thus expected to manipulate more compact memory states [RS01]. However, *under abstraction*, and in particular if there is a structure (which is trackable by the analysis) in the relations between the global-heap and the pending variables, a *global-heap* abstraction in the style of [RS01] might turn out to be more precise than an abstraction of a *local-heap with an unbounded number of cutpoints*.

6.3.3 Interprocedural Shape Analysis using the Functional Approach

Interprocedural shape analysis which uses the functional approach has been studied in [RS01, CR03, JLRS04, HR05, GBC06]. These analyses summarize the behavior of a procedure by tabulating input-output relations. They mainly differ in the representation of the abstract value and in the granularity of the tabulation.

The original motivation for our interprocedural analyses comes from our attempt to obtain a heap-modular interprocedural shape analysis in [Rin01, Chap. 6]. There, this objective was achieved, but based on a weaker technique: (i) a procedure operates on the part of the heap that is reachable from the actual parameters, where the heap is treated as an *undirected* graph; and (ii) pending access paths that point-to objects in the passed part of the heap are represented. In this thesis, the heap is treated as a directed graph and pending access paths are not represented. In addition, [Rin01, Chap. 6] does not handle recursive procedures.

Chong and Rugina [CR03] present a heap-modular interprocedural shape-analysis algorithm. The main idea there is to record for every object both its current properties and the properties it had at the time the procedure was invoked. More specifically, a procedure is analyzed only in the part of the heap that is reachable from its parameters. The algorithm is able to (conservatively) relate sets of abstract objects at the exit memory state with the set of abstract objects that represent the same concrete objects at the entry state by (immutably) labeling *every* abstract object with its properties at the entry state.

Jeannet et al. [JLRS04] present an approach to extend the framework of [SRW02] to handle procedure calls based on two-vocabulary stores. In their approach, procedures are considered as transformers from the (entire) heap before the call, to the (entire) heap after the call. Every heap-allocated object is represented at every program point; on the other hand, only the values of the local variables of the current procedure are represented, which means that the irrelevant parts of the heap are summarized to a few (usually a single) summary nodes during the analysis of an invoked procedure. The relevant objects for the invocation are summarized using a two-vocabulary store. One vocabulary records the current properties of the object. The other vocabulary encodes the properties that the object had when the procedure was invoked. The latter vocabulary allows to match objects at the call-site to objects at the exit-site. Note that this scheme never summarizes together objects that were not summarized together when the procedure was invoked.

By using a two-vocabulary store, the analysis of [JLRS04] is able to achieve a finer grained tabulation and prove, for example, that reversing a list twice restores the original list.[4] However, the mapping is determined by the sharing pattern within the part of the heap that is passed to the procedure, and not by the sharing pattern with the context—which is independent of the internal structure of the local-heap. On the other hand, the approach of [JLRS04] may lead to needlessly large summaries. Consider for example a procedure that operates on several lists and nondeterministically replaces elements between the list tails. The method of [JLRS04] will not summarize list elements that originated from different input lists. Thus, it will generate exponentially more mappings in the procedure summary, than the ones produced by our method. The latter problem is addressed in [JLRS08] by abstracting away some of the fine-grained relational information, an abstraction which greatly improves the scalability of the analysis.

The common aspect of [CR03] and [JLRS04] is that they attempt to abstract the "functional behavior" of the procedure by using a relation which records its effect on the procedure's local-heap. In both approaches, the mapping is determined by the internal properties of the procedure's local-heap (e.g., aliasing and sharing within the procedure's local-heap). However, the sharing patterns with the other parts of the heap are *independent* of the internal structure of the local-heap. In contrast, in our approach we attempt to record these sorts of sharing patterns. These two sorts of information are orthogonal and can be combined.

Hackett and Rugina [HR05] exploit a staged analysis to obtain a relatively scalable interprocedural shape analysis. This approach uses a scalable imprecise pointer-analysis to decompose the heap into a collection of independent locations. The precision of this approach might be limited as it relies on pointer-expressions appearing in the program's text. Its tabulation operates on global-heaps, potentially leading to a low reuse of procedure summaries.

Gotsman et al. [GBC06] describe a heap-modular interprocedural shape analysis that can handle a bounded numbers of cutpoints. The main idea is to treat a bounded number of cutpoint-labels as, essentially, additional parameters: Every procedure can be seen as having k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, their analysis (non-deterministically) binds these additional parameters with references to the cutpoints. The algorithm was instantiated for singly linked lists. Their tabulation algorithm is based on our algorithm, presented in Appendix F.

Yang et al. [YLB+08] present a heap-modular interprocedural shape analysis which, similarly to [GBC06], is based on a domain of separation logic formulae. Their experimental results indicate that the use of local-heaps provides a speedup of $\times 2$–$\times 3$ in the analysis compared to a global-heap analysis. Furthermore, the use of an interprocedural analysis that passes only the reachable portion of the heap was found to be one of the three key reasons for the scalability of their analysis. (The other two key reasons being an efficient join operator and the discard of intermediate states.)

Marron et al. [MHKS08] present a context-sensitive shape analysis which is employed for automatic parallelization of sequential heap manipulating programs. The interprocedural analysis is based on an abstraction of local-heaps with cutpoints. The analysis uses an abstraction of cutpoint-labels which is based on two main ideas: (i) avoid summarizing cutpoints that are generated by the local variables of the *immediate* caller and (ii) abstract all other cutpoints by recording the set of roots of access paths. The analysis also uses liveness information to avoid recording as cutpoints objects that are only pointed to by dead references.

[4] Note that in [CR03], the *objects* at the entry state are related to the objects at the exit state. In [JLRS04], *fields* of objects at the entry state are related to fields of objects at the exit state.

6.3.4 Shape Abstraction of local-heaps

In our interprocedural analysis for cutpoint free programs (and in our modular analysis for dynamically encapsulated programs) we do not devise new shape abstractions. Instead we show how to *lift* existing *intra*procedural shape analyses, *e.g.*, [MYRS05, DOY06, LAIS06], to obtain interprocedural and modular shape analysis algorithms. Specifically, our analyses are parametric in the abstraction of the local-heap.

The common aspect of [MYRS05, DOY06, LAIS06] is that they abstract the heap as *segments* of lists or trees delimited by local variables or by (a bounded number of) shared nodes. The segmented representation records the *reachability-from-variable* information by tracking *dominated-by-(a bounded set of)-variables* information. It is the latter sort of information which is required by our analyses.

The interprocedural shape analysis of [GBC06], which utilizes the domain of [DOY06], also depends on the domination information recorded by the abstraction. Indeed, when the number of cutpoints exceeds a certain (arbitrary but fixed) threshold, the analysis abstracts away all information regarding the cutpoints and turns references to cutpoints into dangling references which the program is not allowed to access.

6.3.5 Heap Modular Analysis

We use the term *heap modular analysis* to denote a particular kind of *heterogeneous abstraction* [YR04]. A heterogeneous abstraction allows different parts of the heap to be abstracted using different degrees of precision at different program points. A heap modular analysis tracks information about the contents of the procedure local-heap and abstracts away (almost) all the information regarding the contents of the other parts of the heap.

The interprocedural shape analysis algorithms presented in Chapters 3 and 4, as well as the analyses of [JLRS04, CR03, HR05, GBC06], compute heap modular procedure summaries, but are not modular, i.e., they analyze whole programs. The analysis of [HR05] tracks properties of single objects. The other algorithms abstract whole local-heaps. The unique aspect of our abstractions is that they maintain information regarding the sharing patterns between the procedure local-heap and the other (irrelevant) parts of the heap.

The trimming abstraction is presented in Chapter 5 for modular analysis. However, it can also be used for achieving a more efficient interprocedural analysis: a trimmed memory state explicitly represents only a part of the local-heap. This suggests possible benefits both in performance and in reuse. Indeed, in [RRYS06] we have applied such an abstraction for shape analysis of parameterized data structures. The heap abstraction of [RRYS06] seeks to strike a balance between the use of non-local (transitive) properties to gain precision and exploiting heap-locality. Similarly to our modular analysis, the abstraction of [RRYS06] represents the heap as an (evolving) tree of heap-components, with only a single heap-component being accessible at any time. The representation is tailored to yield several benefits: (a) It localizes the effect of heap mutation, enabling more efficient processing of heap mutations; (b) The representation is more space-efficient as it permits heap-components with isomorphic contents to use a shared representation; (c) It enables a more precise identification of the "input heap" to a procedure, increasing the reuse of summaries in a tabulation-based interprocedural analysis, thus making it more efficient. More specifically, based on this heap abstraction, we have designed an analysis that can compute parameterized summaries which can be re-used for analyzing clients of instantiations of parameterized (generic) data-structures.

6.4 Modular Shape Analysis

Cousot and Cousot [CC02] describe fundamental techniques for modular static program analysis. These techniques allow to compose separate analyses of different program parts, analyses which detect properties pertaining solely to the analyzed program part, together with a global analysis which detects global program properties. A modular shape analysis, mainly interested in properties of the global-heap, is at risk of degenerating into a whole program analysis. Nevertheless, we are able to achieve modularity by associating (an unbounded number of) different *parts of the heap* (components) with different modules. Our analysis eliminates the need to consider intermodule aliasing by establishing rigid spatial interfaces between parts (components) belonging to different modules and requiring that the heap be, effectively, a tree of components. This allows us to consider properties of components as being module-local and directly use the techniques of [CC02]. In our modular analysis, we use various techniques described in [CC02]:

- We use *user provided interfaces* to prevent live intermodule sharing and to communicate the (limited) effect of mutations done by different modules.
- Our definition of a component ensures that (i) different components never share parts of the heap and that (ii) only component headers can be passed as parameters to intermodule procedure calls. The latter restriction ensures that intermodule procedures are always passed whole components as parameters, and thus, prevents intermodule procedure calls from having side-effects on components not passed as parameters. Every dynamically encapsulated memory state satisfies the above two restrictions. Our analysis utilizes this fact to *simplify* the separate analysis of modules by representing only those parts of the heap which are relevant for the analyzed module.
- Furthermore, we can disregard possible worst case assumptions regarding aliasing in favor of certain benign disjointness assumptions. However, we do make *worst-case assumptions* regarding the possible calling sequences of intermodule procedure calls. An analysis which conservatively abstracts the results of such calling sequences can find all possible heap components of a module because it can treat heap components as atomic values.

Logozzo [Log03] presents a modular analysis which infers class invariants based on an abstraction of program traces. An extension of this work which handles subtyping is presented in [Log04a]. A different extension which allows to consider clients that invoke only certain series of procedure invocations, provided that the language of the expected series can be specified by a regular expression, appears in [Log04b]. The determined invariants concern values of atomic fields of objects of the analyzed class. Properties of subobjects can also be detected provided that these subobjects are encapsulated inside the state of their containing objects (objects which are of the analyzed class) and are never leaked to the context. (A subobject is leaked if it was passed as a parameter from the context or as a return value. For example, in the running example of Chapter 5, resources are "leaked" from resource pools.)

Aggarwal and Randall [AR01] describe an analysis which modularly determines invariants regarding the value of an integer field and the length of an array field of the *same* object. Our analysis computes shape invariants of subheaps comprised of objects that may be passed as parameters.

Yorsh et al. [YSRS05] provide a method for computing the effect of a procedure call which is modular in the program code—but not in the program state: A theorem prover is used to propagate the effect of a procedure call on the different abstraction layers by inferring this effect from the calls effect on the representation of the lower level of abstractions. Intuitively, their approach leads to a breaching of abstraction layers as it requires maintaining in the state of every component information pertaining to the representation of its subcomponents.

Lam et al. [LKR05] and Wies et al. [WKL+06] utilize user-specified pre- and post- conditions to achieve modular shape analysis which can handle a bounded number of flat set-like data structures. It allows objects to be placed in multiple sets. In our approach, an object can be placed only in a single separately-analyzed but arbitrarily-nested set.

One of the most challenging problems in modular analysis is inferring the preconditions of (intermodule) procedure calls. Calcagno et al. [CDOY07] present an analysis which attempts to infer a description of only the memory cells that might be accessed, following the footprint idea in separation logic. The main idea is to try to discover assertions that describe the footprint, rather than the entire global state of the system. Specifically, the analysis runs forward and whenever it encounters a dereference of a potentially dangling pointer, it adds this pointer into the "footprint assertion". The latter describes the cells needed for the program to run safely. Our modular analysis also runs forward. However, it utilizes the (verified) restrictions on the expected "well behaved" clients to compute all possible input states to a procedure (i.e., compute its precondition) using the module's most general client.

6.5 Manual Modular Verification of Heap Manipulating Programs

In this thesis, we do not provide a proof system per-se. However, abstract-interpretation can be seen as a mechanism for automatic program verification [Cou03]. In particular, there are close relations between our approach for interprocedural analysis and existing techniques for manual verification of procedural programs.

6.5.1 Rule of Adaptation

The first proof rule for procedure calls, the *rule of adaptation*, was given in [Hoa71]. It allows to reuse a proof of a procedure body in different invocations of the procedure. Later work, e.g., [ILL74, GL80], simplified the use of this rule by providing a *rule of invariance* (also known as a *frame axiom* or a *frame rule*). The frame axiom enables one to prove that any predicate that does not refer to variables changed by a procedure p (and any procedure invoked by p) can be assumed to remain true after an invocation of p [GL80]. However, heap-manipulating programs are not handled in [Hoa71, ILL74, GL80]. Proof rules for heap-manipulating programs are given in [Hoa72]. These rules are valid only for programs without reference parameters that use tree-like data structures, i.e., programs that do not use sharing.

Our interprocedural shape analysis algorithms are based on an abstraction of a local-heap semantics. Specifically, the "mimicry" of the modular treatment of the heap by the static-analysis algorithms (which is due to the fact that they are abstract interpretations of local semantics) can be seen as a utilization of a (reachability-based) frame rule that is "built into" the semantics. The reuse of the results of an analysis of a procedure-body for different calling-contexts with similar sharing patterns can be seen as a utilization of (a limited form of) an adaptation rule.

6.5.2 Separation Logic

Separation logic [IO01, Rey02] provides a way of proving properties of (heap manipulating) procedures in a local fashion using a *frame rule*. The main idea is to partition the heap into disjoint parts,[5] and reason about each part separately. Inferring the effect of a procedure on a heap described by $P * R$[6] by (only) reasoning about its effect on a heap P is possible, as long as there is no need to reason about the *contents* of the heap described by R. In this case, the frame rule ensures that if a procedure p transforms a heap P into a heap Q, then invoking p on a heap $P * R$ results in a heap that satisfies $Q * R$.

In some sense, the approach used in this chapter is in the spirit of local reasoning. Our semantics resembles the frame rule in the sense that the effect of a procedure call on a large heap can be obtained from its effect on a subheap. However, while the frame rule allows for an arbitrary partitioning of the heap, in our semantics, an invoked procedure operates on the subheap reachable from the actual parameters. In particular, the partitioning of the heap according to the $\mathcal{LSL}^{\text{CPF}}$ semantics and the $\mathcal{LSL}$ semantics is deterministic. (However, in the *analysis*, when the distinction between several cutpoints is lost, the analysis has to take into account every possible matching between the cutpoints at the entry-site and the cutpoints at the exit-site.)

6.5.2.1 Local Reasoning

In separation logic, local reasoning is carried out using user-supplied specifications, e.g., loop invariants and procedure specifications. In contrast, in our work, the partitioning of the heap is built into the concrete semantics, and abstract interpretation is used to establish properties in the absence of user-supplied specifications.

6.5.2.2 Modular Verification

O'Hearn et al. [OYR04] and Bierman and Parkinson [BP05] allow to modularly conduct local reasoning [Rey02] about abstract data structures and abstract data types with inheritance, respectively. The reasoning requires user-specified resource invariants and loop invariants. The model of [BP05] allows for more sharing than our model. However, our modular analysis automatically infers resource invariants based on an ownership transfer specification (and an instance of the bounded parametric abstraction). Technically, our use of rim-objects (resp. abstract sealed components) is analogous to [BP05]'s use of *abstract predicates*' names (resp. resource invariants).

6.5.2.3 Store-based vs. Storeless Semantics

The standard models for separation-logic are based on a store-based semantics [YO02]. In particular, allocation of new heap cells is not parametric because the identity of the location of the allocated cell can be observed in the model [BY07]. As a result, it is very challenging to develop a model for separation logic that can justify *relational*

[5] Mutual references between the different parts of the heap are permitted.

[6] $P * R$ asserts that the heap can be partitioned into two disjoint parts, one satisfying P and the other satisfying R.

parametricity, i.e., a model that can justify observational equivalences between two implementations of a mutable abstract data type [BY07, BY08]. A solution to this problem is given by Birkedal and Yang [BY07, BY08] by using FM domain theory [BL05]. In this domain one can name locations but cannot observe the identity of locations because of the built-in use of permutation of locations. Essentially, the permutation ensures that a program cannot distinguish between isomorphic data structures. This restriction on the observational and computational power of the program allows Benton and Leperchey [BL05] to reason about contextual equivalence [JM96] and Birkedal and Yang [YO02, BY08] to give a parametric model of separation logic, which captures that clients behave parametrically in the internal resource invariants of mutable abstract data types, and in particular, handles the problem of non parametric memory allocation.

In our work, we use storeless semantics which provides a canonical representation for memory states which, in a store-based semantics, would be considered isomorphic. (In particular, memory allocation is deterministic without depending on a particular allocator). This allows our semantics to be fully-abstract (see Lemma 4.5.8 and Theorem 4.5.10). We note that it is much easier to show these properties for our semantics as we only discuss first order languages and do not support higher order functions.

6.6 Encapsulation

Encapsulation (also known as *confinement* or *ownership*) [Hog91, Alm97, CPN98, NVP98, BV99, CNP01, GPV01, MPH01, BN05, LPHZ02, BLS03] allows for modular reasoning about heap-manipulating (object-oriented) programs. The common aspect of these works is that they all place various restrictions on the types of data structures that a program is allowed to manipulate—in particular, on the sharing patterns permitted in the manipulated data structures. See [NBT$^+$03] for a comparison of different models of encapsulation.

Most related to our work are *deep ownership models* which structure the heap into a tree of so-called *owner contexts*. Many contributions to the field use type systems to enforce the structure. For programs that adhere to their restrictions, it can be shown that the restricted structure of the heap ensures certain beneficial properties, e.g., *confinement* of objects to be manipulated only by methods that come from their own packages [BV99], *synchronization* properties can be defined and guaranteed [BLR02], and a certain *frame rule* can be provided [MPHL03].

The cutpoint-freedom restriction (see Section 3.3) is inspired by the restrictions used to achieve encapsulation by using owners-as-dominators. Cutpoint-freedom allows for arbitrary heap sharing within the same procedure, but restricts both the heap sharing and the stack sharing across procedure calls. In a sense, cutpoint-freedom attempts to encapsulate the procedure local-heap in the global context by ensuring that the objects passed as parameters to the procedure dominate its local-heap. This allows to treat the passing of the local-heap to the callee as, essentially, a value-result parameter which is simultaneously assigned to its actual parameters.

The interprocedural shape analysis presented in Chapter 4 does not place *any* restriction on the data structures that the program uses. For example, our list abstraction of the $\mathcal{LSL}$ semantics, presented in Section 4.7, does not place any restriction on the sharing between different lists. However, we expect that the analysis would benefit when analyzing encapsulated programs, because we anticipate that encapsulated programs would have few cutpoints.

The dynamic encapsulation restriction (see Definition 5.5.4) is very similar to the owners-as-dominators restriction. Specifically, our module-induced decomposition of a memory state into a tree of components is similar to the package-induced partitioning of a memory state into a tree of memory-regions in [ZNV04]. Our constraints are also similar to external uniqueness [CW03], which requires that there be a *unique* reference pointing to an object from outside its (transitively) owned context. Such references can be transferred via destructive reads and borrowed within a program scope where their uniqueness may not hold. References going out of components into ancestor owners are allowed.

A *distinguishing aspect of our modular analysis is that it integrates a shape analysis with encapsulation constraints*. More specifically, the dynamic encapsulation restriction imposed in our modular analysis is inspired by the restrictions used to ensure encapsulation. Furthermore, we show that dynamic encapsulation helps modular shape analysis and that shape analysis can be used to verify dynamic encapsulation. Technically, we use shape analysis to establish dynamic encapsulation which allows for arbitrary aliasing within a component and for multiple references from an owner component to the header of an owned component. However, the (live) inter-component references are required to form a tree.

Our use of sealed and unsealed components is similar to the use of packed and unpacked owner contexts in

Boogie [BDF+04, LM06]: In a packed context, class invariants have to hold. Children of packed contexts must also be packed. Modification of objects is only allowed in unpacked contexts. Whereas in our approach sealing is implicitly connected to the semantics of procedure calls, packing and unpacking has to be explicitly specified by the programmer in Boogie, which allows Boogie to handle reentrancy. The central difference between the approaches is that our techniques infer module invariants whereas Boogie verifies class invariants provided by the programmer.

Our ownership specification is also in the spirit of [CW03]'s destructive reads and borrowing. (We note that an interesting opportunity is to try to combine [CW03] with our approach.) Boyland [Boy01] uses shape analysis to modularly verify (specified) uniqueness of a *live* reference to an *object* (which may have live references pointing to its subobjects).

Chapter 7

Conclusions and Future Directions

Our long term research goal is to devise precise and efficient static shape analysis algorithms for proving interesting properties of realistic programs. We believe that this thesis makes an important step towards achieving this goal. In the following, we briefly summarize the main contributions made in this thesis, discuss their strengths and weaknesses, and indicate directions for future work.

We note that the ideas presented in this thesis have been already used in the interprocedural shape analysis algorithms of [GBC06, BFQ07, YLB+08, MHKS08] and in the context-bounded analysis of multithreaded programs of [BFQ07]. (See Section 6.3). They also form the basis for our followup work on modular shape analysis for concurrent libraries [RBR+08]. (See Section 7.3.2).

7.1 Procedure Local-Heap Storeless Semantics

In this thesis, we develop two storeless semantics: $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$. Our semantics are designed for languages with dynamic memory allocation, destructive updating and procedure calls. Our storeless semantics are unique in that called procedures are only passed *parts* of the heap.

The development of a storeless semantics that does not represent all the heap has been challenging. Intuitively, it is hard to develop a storeless semantics which is modular in the heap because memory locations are not explicitly represented. Instead, every dynamically allocated object is represented by the set of pointer-access paths pointing to it. Our main insight in $\mathcal{LSL}$ is that the side-effects of a procedure invocation on pending access paths[1] can be delayed to the procedure return—even though the memory cells do not have unique identifiers (e.g., locations). The main idea is to give special treatment to *cutpoints*: objects that separate the "local-heap" that can be accessed (and possibly mutated) by a procedure from the parts of the heap which it cannot access.

$\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ differ in the way that they treat cutpoints: $\mathcal{LSL}^{\text{CPF}}$ forbids cutpoints. Specifically, it aborts when a procedure invocation yields a cutpoint. $\mathcal{LSL}$ allows arbitrary cutpoints by treating them differently than other, non-cutpoint, objects. More specifically, $\mathcal{LSL}$ labels every cutpoint with the set of access paths pointing to it at the time the invocation starts. This provides a unique canonic context-independent representation for the cutpoints of the invocation. We note that both semantics provide a canonical representation for memory states which, in a store-based semantics, would be considered isomorphic. (In particular, memory allocation is deterministic).

The notion of a *cutpoint* seems to be an important concept both in storeless semantics and in store-based semantics. For instance, garbage collection of local-heaps becomes unsound unless cutpoints are considered as part of the root set. (See Remark 4.3.5 and Corollary 4.5.5).

We characterize the manner in which $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ are equivalent to the standard store-based semantics. (See Section 3.5 and Section 4.5). This allows us to identify a class of assertions for which the non-standard concrete semantics is equivalent to the standard store-based semantics. In addition, we show that $\mathcal{LSL}$ is fully abstract, i.e., whenever two code blocks are indistinguishable in every program context, the two code blocks have the same semantics

[1] We use the term a *pending access path* for an access path which starts at a local variable of a pending call. (See Section 2.4.1.1).

$\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ were designed with their precise and efficient abstractions in mind. Past works [Deu92a, Deu94, Ven99, SRW02] have shown that algorithms based on abstract interpretation of storeless semantics can successfully verify properties such as the absence of null dereferences, the absence of garbage, and preservation of invariants of abstract data types (for small programs). The challenge is to scale up these methods to handle real-life programs. The main concept in our work is to exploit the compositional structure of programs by finding suitable methods for handling procedure calls:

- $\mathcal{LSL}$ distills the information about the irrelevant part of the heap, i.e., the part of the heap which is not reachable from the local variables of the procedure, into its sharing patterns with the procedure's local-heap. The sharing patterns are registered in the form of cutpoint-labels—which are expressible in a context-independent manner—and hence analysis results can be reused for different contexts that have the same sharing patterns. Thus, we believe that an analysis based on an abstraction of our new semantics has better chances to scale both in the size of the analyzed program, and in the precision of the results.
  Furthermore, we believe that $\mathcal{LSL}$ can be used as a formal basis for new static analyses and to formally justify previous analyses that rely on similar observations to ours by showing that these analyses are an abstract interpretation of $\mathcal{LSL}$. Indeed, in Section D.4, we show the first stage of formally justifying previous analyses using $\mathcal{LSL}$: establishing a Galois connection between the concrete program states and the analysis' abstract domain.
- $\mathcal{LSL}^{\text{CPF}}$ was designed with the goal to lift existing intraprocedural analyses to the interprocedural setting. Indeed by restricting the sharing patterns to include only objects that are pointed to by variables, and, in particular, exclude cutpoints, we can lift *intra*procedural analyses of [MYRS05, DOY06, LAIS06] to the interprocedural and modular setting.

One of the design decisions taken when developing $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$ was to replace the (more standard) *call-by-reference* calling convention, where procedures are passed references to the global-heap, with a *call-by-value-result* calling convention, with local-heaps being the value *copied* when the procedure starts executing, and *restored* when it terminates. Thus, it was natural to present our local-heap semantics as large-step semantics [Kah87]. However, the idea of local-heap semantics can be used in small-step semantics [Plo81] too: Instead of encoding a stack of activation records inside the memory state, as traditionally done, it is possible to maintain a *stack of program states*. This stack allows us to represent in every program state the (values of) local variables and the local-heap of just one procedure, as is the case in large-step semantics. This approach is taken in Chapter 5 and in [BFQ07], which presents a context-bounded analysis for multithreaded Java programs using a concrete store-based local-heap semantics.[2] Interestingly, this approach also allows us to apply our techniques to an interesting class of concurrent programs. (See Section 7.3.2).

7.1.1 Limitations

We address the challenging problem of designing shape analysis algorithms following a two-stage approach. In the first stage, we identify an interesting class of programs and an interesting class of properties. We define a non-standard operational semantics, namely $\mathcal{LSL}^{\text{CPF}}$ and $\mathcal{LSL}$, which agrees with the standard semantics with respect to the verified properties in the chosen class of programs. (The semantics also detects whether a program belongs to this class.) In the second phase, we develop abstract interpretation algorithms of our semantics. Thus, by design, our semantics do not preserve all program properties.

A main limitation of our semantics is that they abstract away all properties of the contents of irrelevant parts of the heap. We remark that $\mathcal{LSL}$ and $\mathcal{LSL}^{\text{CPF}}$ preserve the following properties:[3] (i) the values computed by arbitrary code blocks and program expressions; (ii) partial correctness for program properties, in particular, the absence of null-dereferences and the maintenance of data-structure invariants; (iii) infinite executions and total correctness for program properties expressed using the aforementioned assertion language; and (iv) the absence of garbage.

Both $\mathcal{LSL}$ and $\mathcal{LSL}^{\text{CPF}}$ record some, but not all, information about the sharing patterns between the procedure local-heap and the irrelevant context. For example, neither $\mathcal{LSL}$ nor $\mathcal{LSL}^{\text{CPF}}$ preserve the property that an object which is pointed to by a parameter is also pointed to by a field of an object from outside the local-heap. Thus, for example, using $\mathcal{LSL}$ or $\mathcal{LSL}^{\text{CPF}}$ we cannot show that such an object is not heap shared. $\mathcal{LSL}$ does not preserve

[2] The main idea in [BFQ07] is to encode in the local-heaps of the procedures of *every thread* the cutpoints induced by references from *all* threads.

[3] $\mathcal{LSL}^{\text{CPF}}$ preserves these properties only for cutpoint-free programs.

the property that an object is pointed to by a field of an object from outside the local-heap. Thus, it can prove that a cutpoint is not heap shared.

The main limitation of $\mathcal{LSL}^{\text{CPF}}$ is the (intentionally) restricted class of cutpoint free programs which it can handle without aborting in an error state. Unfortunately, we do not have a syntactic characterization of this class of programs, we only have a semantic characterization.

7.1.2 Future Directions

Beyond Reachability. Our local-heap storeless semantics include in the procedure's local-heap all the objects that are reachable from the actual parameters. This design choice was mainly due to pragmatic considerations: Using reachability, we are able to provide a general specification of the procedure's local-heap which, by definition, contains all the objects that a procedure can access. On the flip side, we expect that this specification will usually over-approximate the part of the heap which a procedure actually accesses.[4] In particular, it may unnecessarily generate many cutpoints, which are hard to abstract. Thus, it can be beneficial to devise a storeless semantics which includes in the local-heap of a procedure only a portion of its reachable heap.

A main challenge in devising a semantics which represents only subsets of the reachable heap is the identification of these subsets. One viable approach is to use procedure-specific user-supplied assertions, e.g., using separation logic, about the expected footprint. For example, in [RRYS06], we have already started investigating the use of user specified *pack* and *unpack* assertions to define a semantics that represents a portion of the procedure's local-heap as a basis for shape analysis of parameterized data-structures. (See Section 6.3.5). Another approach, which we have already started to explore, is to define this subset based on certain programming styles and practices. For example, in the trimmed semantics, presented in Chapter 5, the subset is defined based on the module partitioning of the program.[5]

Another challenge in devising a procedure local-heap semantics which operates on a portion of the reachable heap is the need to consider pointers that go from the callee's local-heap into the caller's local-heap. In contrast, the local-heap semantics developed in this thesis had to consider only pointers that go from the caller's local-heap into the callee's (because the callee's local-heap contains all the objects that it can reach). As a result, passing to the callee only a portion of the reachable-heap, complicates the propagation of the effect of the callee to the caller. In particular, it may require a new kind of "forward-going" cutpoints.

Foundations of Storeless Semantics. In a store-based semantics, an object is identified by its location. In a storeless semantics, an object is identified by the set of pointer-access paths pointing to it. An interesting question is how should a semantics which uses locations to represent objects but does not distinguish between isomorphic memory states be classified. On one hand, such a semantics is implemented like a store-based semantics, i.e., using locations. On the other hand, it misses the main characteristic of a store-based semantics, i.e., that an object can be identified in different memory states by its location.

In this thesis, we identified the term "storeless semantics" with semantics which are *implemented* using an equivalence relation over access paths. However, from the point of view of a program, a storeless semantics based on access-paths and a semantics which represents a memory state using locations, à la store-based semantics, but which identifies all isomorphic memory states,[6] à la storeless semantics, are undistinguishable.

We believe that a semantics which uses locations but identifies isomorphic memory states is more closely related to storeless semantics than to store-based semantics. The main advantage of using such a semantics is that it reduces the notational overhead which is imposed by the current "access-path-based" storeless semantics. The main disadvantage is that instead of having a canonic representation for all "isomorphic" memory states, as in

[4] The part of the heap which the procedure actually accesses is known in separation logic terminology [IO01, Rey02] as the *procedure's footprint*.

[5] The trimmings semantics, like the $\mathcal{DOS}$ semantics, is implemented as a store-based semantics using locations. However, this implementation was chosen mostly for simplicity. Specifically, it is possible to define a storeless version of $\mathcal{DOS}$ (and of the trimming semantics) because programs executed in the $\mathcal{DOS}$ semantics cannot observe location names and, in addition, the actual location names of locations in different components do not matter neither in the concrete semantics nor in its abstract interpretation. Specifically, one way to define $\mathcal{DOS}$ as a storeless semantics is to encode the internal structure of a component using the access paths starting at its header; explicitly represent the graph of components; and encode the inaccessible value as a special kind of an atomic value.

[6] This sort of semantics is used in the framework of [SRW02], where the actual identifiers of the individuals in a logical structure do not matter, and in the FM-domain-based semantics for separation logic [BY07, BY08] (see Section 6.5), where locations can be thought of as being (consistently) renamed after every statement.

current storeless semantics, the semantics has to represent a state using a (representative of a) set of isomorphic memory states. Adopting this point of view poses an interesting question: Can we define the difference between a store-based semantics and a storeless semantics based on properties instead of its implementation?

We believe that finding a *declarative definition* of *what* is a storeless (resp. store-based) semantics based on the properties of the semantics instead of its implementation is an interesting and an important problem. Ideally, such a definition would not only characterize the class of storeless (resp. store-based) semantics, but also clarify the differences between storeless and store-based semantics. (See also Section 6.5.2.3).

7.2 Interprocedural Shape Analysis

In this thesis, we present new interprocedural shape-analysis algorithms for programs that manipulate dynamically allocated storage by abstract interpretation of our non-standard local-heap concrete semantics. Our approach is orthogonal and complementary to previous works that analyze procedure invocations [RS01, CR03, JLRS04]. In particular, our approach emphasizes the need to maintain in the analysis information regarding the sharing patterns between the procedure local-heap and the other (irrelevant) parts of the heap. In contrast, existing approaches concentrate on tracking only the internal properties of a procedure's local-heap (e.g., aliasing and sharing within the procedure's local-heap). Intuitively, the importance of explicitly tracking these sharing patterns stems from the fact that they are *independent* of the internal structure of the local-heap. We believe that the modular treatment of the heap will have a key role in scaling interprocedural shape analysis to larger pieces of code.

By choosing to develop static analyses by abstract interpretation of our non-standard semantics, we made two design choices: One is to use a storeless semantics. The other is to concentrate on a superset of a program's footprint, based on reachability, rather than the actual footprint. While the ideas underlying our approach apply also to store-*based* semantics, the choice of a storeless semantics was a natural one to make (see Section 6.1). The decision to concentrate on a superset of a program's footprint (inferable via static analysis), was a pragmatic choice which allows to use the same specification of the footprint for any procedure. (This pragmatic choice was found to be practical by experiments carried out by Yang et. al. [YLB+08]. See Section 6.3.1.2). Nevertheless, in Chapter 5 and in [RRYS06] (see Section 6.3.5), we have already started investigating the use of analyzing a procedure over a subset of its reachable heap utilizing light-weight user-supplied specifications.

The notion of a *cutpoint* seems to be an important concept in interprocedural shape analysis algorithms, independently of the concrete semantics on which the analysis is based. Specifically, one has to devise ways either to specialize the algorithms to handle programs with restricted sharing patterns which bound the number of cutpoints, or to devise methods to abstract (an unbounded number of) cutpoints. We follow both methods:

- We define the restricted class of cutpoint-free programs. We show that interesting cutpoint-free programs can be written naturally, e.g., programs manipulating unshared trees and a recursive implementation of quicksort. (We also show that some interesting existing programs are cutpoint-free, e.g., all programs verified using shape analysis in [DRS00, RS01, JLRS04], and many of those in [Deu94, SYKS03].)
- We present an interprocedural analysis which is precise enough for programs with a small number of cutpoints.

Our analyses are modular in the heap (see Section 6.3.5) and thus allow reusing the effect of a procedure at different calling contexts and at different call-sites. Our analysis goes beyond the limits of existing approaches and was used to verify a recursive quicksort implementation. Preliminary experimental results indicate that:

- the cost of analyzing recursive procedures is similar to the cost of analyzing their iterative versions;
- our analysis benefits from procedural abstraction—often the handling of procedures is as precise as inlining the procedure body and the cost can be smaller when the code is reused. In fact, in some cases the precision of the analysis can be improved by procedural abstraction. In particular, the analysis of recursive procedures can be more efficient or even more precise than the analysis of the iterative version; and
- our approach compares favorably with [RS01, JLRS04].

7.2.1 Limitations

Our current analyses are expected to become imprecise when the abstractions summarize together multiple cutpoints. In particular, analyzing programs that generate an unbounded number of cutpoints can be very challenging

for our analyses. Devising abstractions suitable for handling such programs is a matter of future work. (See Section 7.2.2).

Our interprocedural shape analyses require maintaining a rather precise information regarding reachability and domination in the heap. This forces our analyses to use very precise *intra*procedural analyses which are capable of keeping track of this sort of information. (See Section 6.3.4).

7.2.2 Future Directions

Detecting Live Cutpoints. Our interprocedural shape analysis for cutpoint-free programs aborts the program when a cutpoint is detected. However, if a program never uses the references which generated the cutpoints, this restriction seems needlessly severe, as the generated cutpoint is, in a sense, a *dead* cutpoint. In the future, we plan to utilize liveness analysis or, alternatively, user-supplied specification to separate the live cutpoints from the dead cutpoints and abstract dead cutpoints as regular, i.e., non-cutpoint, objects.

Cutpoint Abstraction. The cutpoint abstraction becomes challenging when there is no a priori bound on the number of cutpoints. In these programs, it is possible to obtain bounded abstractions only when several cutpoint objects are merged. The need to "merge" objects to obtain bounded abstraction is not new. In the past, analyses that were developed within the 3-valued logical framework for program analysis of [LAS00, SRW02] addressed it by merging together objects that have "similar" aliases. In this thesis, we suggested abstracting cutpoints in a similar way: e.g., merge together cutpoints that have the same type or that are reachable from the same formal variables on procedure entry. (See Section 4.10.4.1). However, it is not clear if this is the right way to go. We believe that devising abstractions of cutpoints which are useful for analyzing real-life programs is an interesting and important research problem.

Cutpoint Profiling. A first stage in developing *useful* abstractions for cutpoints can be an experimental study of the sorts of cutpoints and sharing patterns that arise in real-life programs. Rubinstein [Rub06] provides a cutpoint-profiler which measures the number of cutpoints that occur during executions of real-life programs. The study determined that there are methods with a very large number of cutpoints. It also detected that most of the cutpoints stem from sharing of `String` objects, `Integer` objects, and other objects of similar types. Thus, it seems that an analysis should take special care of this kind of cutpoints. The findings of [Rub06] strengthen the need to refine the measurements in order to gain more intuition that can help in finding cutpoint abstractions which are useful for analyzing real-life programs. In particular, it is interesting to measure cutpoint liveness information.

Scaling local-heap Shape Analysis. There are several techniques that can be combined with local-heaps to scale our shape analysis. For example, *staged analysis* can be used to reduce the cost of abstraction by first computing context-sensitive flow-insensitive points-to analysis (e.g., [WL04]) and then employing our fine-grained abstraction on parts of the heap which may be aliased. Another possibility is to follow [YR04], where it is shown how to radically improve the efficiency of shape analysis by using heterogenous abstractions.

Our prototype implementation already benefits from the *partially disjunctive join operator* of Manevich et. al. [MSRF04], which merges similar shape-graphs. This reduces the number of shape-graphs and the running time. Technically, this join operator [MSRF04] exploits the fact that our abstract domain ($2^{3Struct}$) has a Hoare order and returns an upper approximation of the set-union operator. As a result, the analyses may provide a less precise information than the meet-over-all-valid-paths [SP81], which a fully-disjunctive join operator (i.e., set-union) would guarantee. However in our experiments we found that using partially disjunctive join is precise enough and provides for a much more efficient analysis. In recent work, Manevich et. al. [MBC$^+$07] suggested the use of an even more aggressive join operator which is based on heap decomposition. It is interesting to consider developing interprocedural analyses which use this "more aggressive" join operator.

7.3 Modular Shape Analysis

Modular shape analysis is particularly difficult because of aliasing which, in general, is not constrained. In this thesis, instead of trying to devise an analysis for arbitrary programs, we focus our attention to certain "well-behaved" programs.

The main idea behind our approach is to assume a modularly-checkable program-invariant concerning aliases of live intermodule references. More specifically, we focus on *dynamically encapsulated programs*, programs which adhere to the restriction that live references between *components* (subheaps manipulated by different modules) form a tree. We consider our work as a first step towards a modular shape analysis. We note that our modular shape analysis is the first one capable of handling pointer parameters.

A distinguishing aspect of our work is that we integrate a shape analysis with (dynamic) encapsulation constraints. Our work presents a nice interplay between encapsulation and modular shape analysis: it uses dynamic encapsulation to enable modular shape analysis, and uses shape analysis to determine that the program is dynamically encapsulated.

While the dynamic encapsulation model is fairly restrictive with respect to the coupling between separate components, it is very permissive about what can happen inside a single component. This model is also sufficient to express several, natural, usage constraints that arise in practice. Furthermore, we believe that our restrictions can be relaxed to help address a larger class of programs.

7.3.1 Limitations

The main limitations of our approach is the need for a user provided owner-ship specification, and the restrictions of the inter-component references.

7.3.2 Future Directions

Inferring Ownership Transfer Specification. Our modular analysis requires user-supplied specification regarding ownership transfer. While it is possible to relieve the need for user-supplied specification by fixing arbitrary ownership transfer specification, the resulting analysis, in many cases, may not be useful.

Automatically determining the ownership transfer specification will further enhance the automation of our analysis. One way to find a plausible specification can be to "train" the analysis by performing whole-program analyses of example programs. Another way would be to use dynamic (runtime) analysis.

Partitioned Module Invariants. We use a very conservative abstraction of sealed components and inter-component references. In particular, the abstraction retains no information about the state of a sealed component (which typically belongs to other modules used by the analyzed module). This can lead to an undesirable loss in precision in the analysis (in general). We can refine the abstraction by using *component-digests* [RRSY06], which encode (hierarchical) properties of whole *components* in a typestate-like manner [SY86]. This, e.g., can allow our analysis to distinguish between a reference to a pool of closed socket components from a reference to a pool of connected socket components.

Modular Analysis with Shared Abstractions. Modular verification of shared data structures is a challenging problem: Side-effects in one module that are observable in another module make it hard to analyze each module separately. Our modular analysis forbids (live) intermodule sharing. Juhasz et. al. [Juh08, JRPH+08] present a novel approach for relieving this restriction, thus allowing modular verification of shared data structures. The main idea is to verify that the inter-module sharing is restricted to a user-provided specification which also enables the analysis to handle side-effects.

Modular Local Shape Analysis for Concurrent View-Serializable Programs. In this thesis we focused exclusively on sequential programs. In our current work, we address the challenging problem of modular analysis for *concurrent* programs. Specifically, we aim at designing modular static analysis algorithms for a subset of *view-serializable* heap-manipulating programs. Our main observation is that an assertion, from a certain class of program assertions, holds in all concurrent executions of a *view-serializable* program *if and only if* it holds in all of its serial executions. Furthermore, this class includes assertions that are sufficient to show view-serializability. Specifically, we provide a reduction which allows to transform certain concurrent programs into sequential programs, and by analyzing the resulting sequential programs using sequential techniques, to establish assertions of the concurrent programs. An immediate consequence of our current work, is that it shows how to lift the results presented in this thesis to the concurrent setting.

Bibliography

[Alm97] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 32–59, 1997.

[AR01] A. Aggarwal and K.H. Randall. Related field analysis. In *Programming Languages Design and Implementation (PLDI)*, pages 214–220, 2001.

[AW93] U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82, Washington, DC, September 1993. IEEE Press.

[Bar77] J. M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM (CACM)*, 20(7):513–518, 1977.

[BC05] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'REILLY, 3rd edition, 2005.

[BDF+04] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[Ber79] V. A. Berzins. *Abstract model specifications for data abstractions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1979.

[BFQ07] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *Computer Aided Verification (CAV)*, pages 207–220, 2007.

[BIL03] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 55–65. ACM Press, 2003.

[BL05] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, pages 86–101, 2005.

[BLR02] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 211–230, 2002.

[BLS03] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223, 2003.

[BN05] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005.

[Boy01] J. Boyland. Alias burying: unique variables without destructive reads. *Software: Practice and Experience (SPE)*, 31(6):533–553, 2001.

[BP05] G. Bierman and M. Parkinson. Separation logic and abstractions. In *Principles of Programming Languages (POPL)*, pages 247–258, 2005.

[BR01] T. Ball and S.K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 97–103, 2001.

[BV99] B. Bokowski and J. Vitek. Confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 82–96, 1999.

[BY07] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *FoSSaCS*, pages 93–107, 2007.

[BY08] L. Birkedal and H. Yang. Relational parametricity and separation logic. *Logical Methods in Computer Science (LMCS)*, 4(2:6):1–27, May 2008.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Programming Languages Design and Implementation (PLDI)*, pages 47–56, New York, NY, 1988. ACM Press.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CC78] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, New York, NY, 1979. ACM Press.

[CC02] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *Compiler Construction (CC)*, pages 159–178, 2002.

[CDOY07] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, pages 402–418, 2007.

[CNP01] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 53–76, 2001.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM (CACM)*, 13(6):377–387, 1970.

[Cou96] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.

[Cou97] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science (TCS)*, 6:47–103, 1997.

[Cou00] P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 224 and electronic proceedings *http://www.ssgrr.it/en/ssgrr2000/proceedings.htm*, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.

[Cou03] P. Cousot. Automatic verification by abstract interpretation, invited tutorial. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 20–24. LNCS 2575, 2003.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 48–64, 1998.

[CR03] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *Static Analysis Symposium (SAS)*, pages 310–323. ACM, 2003.

[CW03] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 176–200, 2003.

[CWZ90] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Programming Languages Design and Implementation (PLDI)*, pages 296–310, New York, NY, 1990. ACM Press.

[Deu92a] A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, LIX, Ecole Polytechnique, F-91128, Palaiseau, France, 1992.

[Deu92b] A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, Washington, DC, 1992. IEEE Press.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Programming Languages Design and Implementation (PLDI)*, pages 230–241, 1994.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Programming Languages Design and Implementation (PLDI)*, pages 57–68, 2002.

[DOY06] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 287–302, 2006.

[DRS00] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symposium (SAS)*, pages 115–134. Springer, 2000.

[GBC06] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium (SAS)*, pages 240–260, 2006.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[GL80] D. Gries and G. Levin. Assignment and procedure call proof rules. *Transactions on Programming Languages and Systems (TOPLAS)*, pages 564–579, October 1980.

[GPV01] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 241–253, 2001.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM (CACM)*, 4(7):321, 1961.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. *Lecture Notes in Mathematics*, 188:102–116, 1971.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hog91] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 271–285, 1991.

[HR05] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Principles of Programming Languages (POPL)*, pages 310–323, 2005.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 104–115, 1995.

[ILL74] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification I: A logical basis and its implementation. *Acta Informatica*, 4:145 – 182, 1974.

[Imm99] N. Immerman. *Descriptive Complexity*. Graduate Texts in Computer Science. Springer, New York, 1999.

[IO01] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages (POPL)*, pages 14–26, 2001.

[JLRS04] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium (SAS)*, pages 246–264, 2004.

[JLRS08] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. (Submitted for journal publication), 2008.

[JM81] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[JM82] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Principles of Programming Languages (POPL)*, pages 66–74, New York, NY, 1982. ACM Press.

[JM96] Trevor Jim and Albert R. Meyer. Full abstraction and the context lemma. *SIAM Journal on Computing*, 25(3):663–696, 1996.

[Jon81] H.B.M. Jonkers. Abstract storage structures. In de Bakker and van Vllet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.

[JRPH+08] U. Juhasz, N. Rinetzky, A. Poetzsch-Heffter, M. Sagiv, and E. Yahav. Modular verification with shared abstractions. Tech. rep. (in preperation), Tel Aviv University, 2008.

[Juh08] U. Juhasz. Modular verification with shared abstractions. Master's thesis, Tel-Aviv University, School of Computer Science, Tel-Aviv, Israel (in preperation), 2008.

[Kah87] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39. Springer-Verlag, 1987.

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages (POPL)*, pages 194–206, New York, NY, 1973. ACM Press.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Compiler Construction (CC)*, pages 125–140, 1992.

[LAIS06] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Computer Aided Verification (CAV)*, pages 547–561, 2006.

[LARSW00] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 26–38, 2000.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *Static Analysis Symposium (SAS)*, pages 280–301. Springer, 2000. http://www.math.tau.ac.il/∼ tvla.

[LASIR07] Tal Lev-Ami, Mooly Sagiv, Neil Immerman, and Thomas W. Reps. Constructing specialized shape analyses for uniform change. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 215–233. LNCS, 2007.

[LH88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *Programming Languages Design and Implementation (PLDI)*, pages 21–34, New York, NY, 1988. ACM Press.

[LKR05] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency (tool demo). In *Compiler Construction (CC)*, pages 237–241, 2005.

[LM06] K.R.M. Leino and P. Müller. A verification methodology for model fields. In *European Symposium on Programming (ESOP)*, pages 115–130, 2006.

[Log03] F. Logozzo. Class-level modular analysis for object oriented languages. In *Static Analysis Symposium (SAS)*, pages 37–54, 2003.

[Log04a] F. Logozzo. Automatic inference of class invariants. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 355–382, 2004.

[Log04b] Francesco Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, pages 332–346. Springer-Verlag, 2004.

[LPHZ02] K.R.M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Languages Design and Implementation (PLDI)*, pages 246–257. ACM Press, 2002.

[LR91] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *Principles of Programming Languages (POPL)*, pages 93–103, New York, NY, January 1991. ACM Press.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Programming Languages Design and Implementation (PLDI)*, pages 235–248. ACM Press, 1992.

[Mar99] F. Martin. Experimental comparison of call string and functional approaches to interprocedural analysis. In *Compiler Construction (CC)*, pages 63–75, 1999.

[MBC+07] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–18. Springer, 2007.

[MHKS08] M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *Compiler Construction (CC)*, pages 245–259, 2008.

[MN99] K. Mehlhorn and S. Naher. *LEDA, A Platform for Combinatorical and Generic Computing*. Cambridge University Press, first edition, 1999.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[MPHL03] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.

[MS77] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Halsted Press, New York, NY, USA, 1977.

[MSRF04] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Static Analysis Symposium (SAS)*, pages 265–279, 2004.

[MYRS05] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 181–198, 2005.

[NBT+03] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, pages 73–87, 2003.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NVP98] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 158–185, 1998.

[OYR04] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *Principles of Programming Languages (POPL)*, pages 268–280, 2004.

[PCK93] J. Plevyak, A.A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 37–57. Springer-Verlag, 1993.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[RBR+04] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local-heaps and its abstractions. Tech. Rep. 1, AVACS, October 2004. Available at "*http://www.avacs.org*".

[RBR+05] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local-heaps and its abstractions. In *Principles of Programming Languages (POPL)*, pages 296–309, 2005.

[RBR+08] N. Rinetzky, A. Bouajjani, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for serializable programs. Tech. rep. (in preperation), Tel Aviv University, April 2008.

[Rey68] J.C. Reynolds. Automatic computation of data set definitions. In *Information Processing 68: Proceedings of the IFIP Congress*, pages 456–461, New York, NY, 1968. North-Holland.

[Rey02] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages (POPL)*, pages 49–61, New York, NY, 1995. ACM.

[Rin01] N. Rinetzky. Interprocedural shape analysis. Master's thesis, Technion, Israel, 2001.

[RPHR+06] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. Tech. Rep. 107, Tel Aviv University, December 2006. Available from $http://www.cs.tau.ac.il/\sim maon$ (technical report TAU-CS-107/06).

[RPHR+07] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *European Symposium on Programming (ESOP)*, pages 220–236, 2007.

[RRSY06] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. Componentized heap abstractions. Tech. Rep. 164, Tel Aviv University, December 2006.

[RRYS06] N. Rinetzky, G. Ramalingam, E. Yahav, and M. Sagiv. Componentized heap abstraction. Tech. Rep. 301, Tel Aviv University, April 2006.

[RS01] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Compiler Construction (CC)*, pages 133–149, 2001.

[RSL03] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming (ESOP)*, volume 2618, pages 380–398, 2003.

[RSW04] Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In *Computer Aided Verification (CAV)*, pages 15–30, 2004.

[RSY04] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local-heaps. Tech. Rep. 26, Tel Aviv Uni., November 2004. Available at "*http://www.cs.tau.ac.il/∼maon*".

[RSY05a] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Static Analysis Symposium (SAS)*, pages 284–302, 2005.

[RSY05b] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. Tech. Rep. 104/05, Tel Aviv Uni., April 2005. Available at "*http://www.math.tau.ac.il/∼maon*".

[Rub06] S. Rubinstein. On the utility of cutpoints for monitoring program execution. Master's thesis, Tel-Aviv University, School of Computer Science, Tel-Aviv, Israel, 2006.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[SRH96] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science (TCS)*, 167:131–170, 1996.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, January 1998.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[SY86] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, 1986.

[SYKS03] R. Shaham, E. Yahav, E.K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Static Analysis Symposium (SAS)*, pages 483–503, 2003.

[SYKS05] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. *Science of Computer Programming (SCP)*, 58(1-2):264–289, 2005.

[Ven99] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming (SCP)*, 35(2):223–248, 1999.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java optimization framework. In *Proceedings of CAS Conference (CASCON)*, pages 125–135, 1999.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[WKL+06] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 157–173, 2006.

[WL04] J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Languages Design and Implementation (PLDI)*, pages 131–144. ACM Press, 2004.

[Yah01] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Principles of Programming Languages (POPL)*, pages 27–40, January 2001.

[YLB+08] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification (CAV)*, pages 385–398. Springer-Verlag, 2008.

[YO02] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 402–416. Springer-Verlag, 2002.

[YR04] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Principles of Programming Languages (POPL)*, pages 25–34. ACM Press, 2004.

[YSRS05] G. Yorsh, A. Skidanov, T. W. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manipulating programs: Ongoing work. *Electronic Notes in Theoretical Computer Science*, 131:125–138, May 2005.

[ZNV04] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 241–251, 2004.

Appendix A

Mathematical Conventions

A.1 Mathematical Notations

In this section, we provide some standard mathematical definitions regarding operations on sets and functions, binary relations, equivalence relations, and sequences.

Definition A.1.1 (Set difference) *Given two sets S_1 and S_2, the **set difference** between S_1 and S_2, denoted by $S_1 \setminus S_2 = \{e \in S_1 \mid e \notin S_2\}$, is the set of elements in S_1 which are* not *also in S_2.*

Definition A.1.2 (Function restriction) *Given a function $f\colon S \hookrightarrow T$ and a set $S_{sub} \subseteq S$, the **restriction of** $\mathbf{f}$ **to** $\mathbf{S}_{sub}$, denoted by*

$$f|_{S_{sub}} = \begin{cases} f(e) & e \in S_{sub} \\ \textit{undefined} & \textit{otherwise} \end{cases},$$

is a function which identifies with f on S' and is not defined anywhere else.

Definition A.1.3 (Least fixed point) *Given a lattice $(D, \sqsubseteq)$ and a function $f(X)\colon D \to D$ which is monotone with respect to $\sqsubseteq$, the **least fix-point of** $\mathbf{f}$, denoted by $lfp\ \lambda.(f)$, is an element $d \in D$ which is (i) a fixpoint of f, i.e., $d = f(d)$, and (ii) smaller than any other fixpoint of f, i.e., if $d' = f(d')$ then $d \sqsubseteq d'$.*

Definition A.1.4 (Binary relations) *A set $\tau \subseteq \Sigma \times \Sigma$ is a **binary relation** over a set Σ. The **domain** of τ, denoted by, $dom(\tau)$, is $dom(\tau) = \{\sigma \mid \langle \sigma, \sigma' \rangle \in \tau\}$. The **range** of τ, denoted by, $range(\tau)$, is $range(\tau) = \{\sigma' \mid \langle \sigma, \sigma' \rangle \in \tau\}$. The **image** of a set $S \subseteq \Sigma$ under τ, denoted by $\tau(S)$ is $\tau(S) = \{\sigma' \in \Sigma \mid \sigma \in S,\ \langle \sigma, \sigma' \rangle \in \tau\}$. The **composition** of τ with a binary relations τ' over Σ, denoted by $\tau \circ \tau'$, is $\tau \circ \tau' = \{\langle \sigma, \sigma'' \rangle \mid \langle \sigma, \sigma' \rangle \in \tau, \langle \sigma', \sigma'' \rangle \in \tau'\}$.*

By abuse of notation, we sometimes denote the image of a singleton set $S = \{\sigma\}$ under a binary relation $\tau \subseteq \Sigma \times \Sigma$ by $\tau(\sigma)$.

Definition A.1.5 (Equivalence relations) *A binary relation $\approx$ over a set Σ is an **equivalence relation** if $\approx$ is* reflexive, *i.e., for every $\sigma \in \Sigma$, $\sigma \approx \sigma$;* symmetric, *i.e., for every $\sigma_1, \sigma_2 \in \Sigma$, $\sigma_1 \approx \sigma_2$ iff $\sigma_2 \approx \sigma_1$; and* transitive, *i.e., for every $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$, if $\sigma_1 \approx \sigma_2$ and $\sigma_2 \approx \sigma_3$, then $\sigma_1 \approx \sigma_3$. The **equivalence class** of $\sigma \in \Sigma$ under an equivalence class $\approx$, denoted by $[\sigma]_{\approx}$, is $[\sigma]_{\approx} = \{\sigma' \in \Sigma \mid \sigma' \approx \sigma\}$. The **quotient set** of an equivalence class $\approx$, denoted by $\Sigma/\approx$ is $\Sigma/\approx = \{[\sigma]_{\approx} \mid \sigma \in \Sigma\}$.*

Definition A.1.6 (Set partitioning) *A set $\mathcal{P}$ is a **partitioning** of a set S if $\mathcal{P} \subseteq 2^S$, $\{s \in p \mid p \in \mathcal{P}\} = S$ and for every $p_1, p_2 \in \mathcal{P}$ such that $p_1 \neq p_2$, $p_1 \cap p_2 = \emptyset$.*

Definition A.1.7 (Sequences) *A **sequence** π over a set S is a total function $\pi \in \{i \in \mathbb{N} \mid 0 \leq i \leq n\} \to S$ for some $n \in \mathbb{N}$. The **length of a sequence** π, denoted by $|\pi|$, is $|dom(\pi)|$. A sequence π is a **subsequence** of a sequence π' if there exists $n \in \mathbb{N}$ such that for any $j \in dom(\pi)$ it holds that $\pi(j) = \pi'(j + n)$; such a π is (also) a **prefix** of π' if $n = 0$. The **concatenation** of sequences π_1 and π_2, denoted by juxtaposition $\pi_1\pi_2$, is a sequence π such that $\pi(i) = \pi_1(i)$ for every i, $0 \leq i < |\pi_1|$ and $\pi(i) = \pi_2(i - |\pi_1|)$ for every i, $|\pi_1| \leq i$.*

$\wedge$	0	1	$\frac{1}{2}$
0	0	0	0
1	0	1	$\frac{1}{2}$
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$

$\vee$	0	1	$\frac{1}{2}$
0	0	1	$\frac{1}{2}$
1	1	1	1
$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$

$\neg$	
0	1
1	0
$\frac{1}{2}$	$\frac{1}{2}$

Figure A.1: Kleene's 3-valued interpretation of the propositional operators.

A.2 Syntax and Semantics of Logical Formulae

In this section , we define the syntax and the semantics of first-order formulae with equality and transitive closure we use.

A.2.1 Syntax of Formulae

In this section , we define the syntax of first-order formulae with equality and transitive closure we use.

Definition A.2.1 *A* **formula** *over the* **vocabulary** $\mathcal{P} = \{p_1, \ldots, p_n\}$ *is defined inductively, as follows:*

Atomic Formulae *The* **logical literals** $\mathbf{0}$, $\mathbf{1}$, *and* $\mathbf{\frac{1}{2}}$ *are atomic formulae with no free variables.*

For every predicate symbol $p \in \mathcal{P}$ of arity k, $p(v_1, \ldots, v_k)$ is an atomic formula with free variables $\{v_1, \ldots, v_k\}$.

The formula $(v_1 \pi v_2)$, where v_1 and v_2 are distinct variables, is an atomic formula with free variables $\{v_1, v_2\}$.

Logical Connectives *If φ_1, φ_2 and φ_3 are formulae whose sets of free variables are V_1, V_2, and V_3 , respectively, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\neg\varphi_1)$, and $(\varphi_1 ? \varphi_2 : \varphi_3)$ are formulae with free variables $V_1 \cup V_2$, $V_1 \cup V_2$, V_1, and $V_1 \cup V_2 \cup V_3$ respectively.*

Quantifiers *If φ is a formula with free variables $\{v_1, v_2, \ldots, v_k\}$, then $(\exists v_1 : \varphi)$ and $(\forall v_1 : \varphi)$ are both formulae with free variables $\{v_2, v_3, \ldots, v_k\}$.*

Transitive Closure *If φ is a formula with free variables V such that $v_3, v_4 \notin V$, then $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$ is a formula with free variables $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$. We use the notations $n^+(v_3, v_4) = (TC\ v_1, v_2 : n)(v_3, v_4)$ and $n^*(v_3, v_4) = (TC\ v_1, v_2 : \varphi)(v_3, v_4) \vee eq(v_3, v_4)$ as shorthand for the transitive closure and the reflexive transitive closure of the binary predicate n, respectively.*

A formula is **closed** *when it has no free variables.*

A.2.2 Kleene's 3-Valued Semantics

Kleene's Interpretation of the propositional operators is given in Figure A.1.

Kleene's 3-valued semantics for first-order formulae with transitive closure is given in Definition A.2.2.

Definition A.2.2 *A* 3**-valued interpretation** *of the language of formulae over $\mathcal{P}$ is a* 3**-valued logical structure** *S, comprised of U^S, a set of* **individuals** *and interpretation ι^S which maps every predicate symbol p of arity k to a truth-valued function:*

$$\iota^S(p) \colon (U^S)^k \to \{0, 1, \frac{1}{2}\}.$$

An **assignment** *Z is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z \colon \{v_1, v_2, \ldots\} \to U^S$). An assignment that is defined on all free variables of a formula φ is called* **complete** *for φ. In the sequel, we assume that every assignment Z that arises in connection with the discussion of some formula φ is complete for φ.*

The **meaning** *of a formula φ, denoted by $[\![\varphi]\!]_3^S(Z)$, yields a truth value in $\{0, 1, \frac{1}{2}\}$. The meaning of φ is defined inductively as follows:*

Atomic *For a logical literal* $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}, \mathbf{\frac{1}{2}}\}$, $[\![\boldsymbol{l}]\!]^S_3(Z) = l$ *(where* $l \in \{0, 1, \frac{1}{2}\}$*).*

For an atomic formula $p(v_1, \ldots, v_k)$*,*

$$[\![p(v_1, \ldots, v_k)]\!]^S_3(Z) = \iota^S(p)(Z(v_1), \ldots, Z(v_k))$$

Logical Connectives *For logical formulae* φ_1*,* φ_2*, and* φ_3

$$\begin{array}{rcl}
[\![\varphi_1 \wedge \varphi_2]\!]^S_3(Z) & = & \min([\![\varphi_1]\!]^S_3(Z), [\![\varphi_2]\!]^S_3(Z)) \\
[\![\varphi_1 \vee \varphi_2]\!]^S_3(Z) & = & \max([\![\varphi_1]\!]^S_3(Z), [\![\varphi_2]\!]^S_3(Z)) \\
[\![\neg\varphi_1]\!]^S_3(Z) & = & 1 - [\![\varphi_1]\!]^S_3(Z) \\
[\![\varphi_1 ? \varphi_2 : \varphi_3]\!]^S_3(Z) & = & \begin{cases}
[\![\varphi_2]\!]^S_3(Z) & [\![\varphi_1]\!]^S_3(Z) = 0 \\
[\![\varphi_3]\!]^S_3(Z) & [\![\varphi_1]\!]^S_3(Z) = 1 \\
[\![\varphi_2]\!]^S_3(Z) & [\![\varphi_1]\!]^S_3(Z) = \frac{1}{2} \\
 & [\![\varphi_2]\!]^S_3(Z) = [\![\varphi_3]\!]^S_3(Z) \\
\frac{1}{2} & otherwise
\end{cases}
\end{array}$$

Quantifiers *If* φ *is a logical formula,*

$$\begin{array}{rcl}
[\![\forall v_1 : \varphi]\!]^S_3(Z) & = & \min_{u \in U^S} [\![\varphi_1]\!]^S_3(Z[v_1 \mapsto u]) \\
[\![\exists v_1 : \varphi]\!]^S_3(Z) & = & \max_{u \in U^S} [\![\varphi_1]\!]^S_3(Z[v_1 \mapsto u])
\end{array}$$

Transitive Closure *(Transitive closure using a pair of variables). For* $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$*,*

$$[\![(TC\ v_1, v_2 : \varphi)(v_3, v_4)]\!]^S_3(Z) = \max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \min_{i=1}^{n} [\![\varphi]\!]^S_3(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

Extended Transitive Closure *(Transitive closure using a pair of pairs of variables. See also Section D.3.1.2).*

For $(TC\ v_1, v_2; w_1, w_2 : \varphi)(v_3, v_4; w_3, w_4)$*,*

$$[\![(TC\ v_1, v_2; w_1, w_2 : \varphi)(v_3, v_4; w_3, w_4)]\!]^{\langle U,\iota\rangle}_2(Z) \stackrel{\text{def}}{=} \max_{\substack{n \in \mathbb{N}, \\ u^1_1, \ldots, u^1_{n+1} \in U, \\ Z(v_3) = u^1_1, Z(v_4) = u^1_{n+1}, \\ u^2_1, \ldots, u^2_{n+1} \in U, \\ Z(w_3) = u^2_1, Z(w_4) = u^2_{n+1}}} \min_{i=1}^{n} [\![\varphi]\!]^{\langle U,\iota\rangle}_2(Z\left[\begin{array}{l} v_1 \mapsto u^1_i, v_2 \mapsto u^1_{i+1}, \\ w_1 \mapsto u^2_i, w_2 \mapsto u^2_{i+1} \end{array}\right])$$

We say that S *and* Z **potentially satisfy** φ *(denoted by* $S, Z \models \varphi$*) if* $[\![\varphi]\!]^S_3(Z) = \frac{1}{2}$ *or* $[\![\varphi]\!]^S_3(Z) = 1$*. Finally, we write* $S \models \varphi$ *if for every* Z*:* $S, Z \models \varphi$*.*

We remind the reader, that we use the equality infix predicate ($=$) in our formula as an intuitive replacement of the eq predicate. In particular, while the eq predicate is defined using the semantics "identically-equal" relation on individuals, denoted also by the symbol '$=$',[1] the two may not denote the same relation:

- Equal elements (according to the eq predicate) are identical, i.e., if $eq(u_1, u_2)$ then $u_1 = u_2$. (In this case u_1 is a non-summary individual.)
- Non-identical individuals u_1 and u_2 are unequal (i.e., if $u_1 \neq u_2$ then $\neg eq(u_1, u_2)$). However, while obviously $u = u$, it can be the case that $eq(u, u)$ evaluates to $\frac{1}{2}$. (In this case, u is a summary individual.)

Notice that Definition A.2.2 could be generalized to allow many-sorted sets of individuals. This would be useful for modeling heap cells of different types; however, to simplify the presentation, we have chosen not to introduce this mechanism.

[1] Throughout the thesis, the intended meaning of the $=$ should always be clear from the context.

Appendix B

Appendix for Chapter 2

B.1 The Meaning of Control Statements

In our analyses, we represent procedures using control-flow graphs and encode conditionals and while loops using `assume` statements. (See Appendix F). An `assume`(*cnd*) statement acts as an identity statement when applied to a state in which *cnd* holds, and gets the computation stuck, otherwise.

Computing whether a condition *cnd* $\in$ *cond* (see Figure 2.1) holds in a a given state is done using a semantics-specific function $\mathcal{B}_{\mathfrak{S}} : \mathit{cond} \to \mathfrak{S} \to \{tt, ff\}$ which maps conditional *cnd* $\in$ *cond* (see Figure 2.1) to predicates over states $\mathfrak{s} \in \mathfrak{S}$. Thus, in the following we only provide such a function $\mathcal{B}_{\mathfrak{S}}[\![\ ]\!]$ for every semantics.

For example, the semantics function which maps conditional involving comparison between access paths α and β to predicates over store-based states is $\lambda s_G.[\![\alpha = \beta]\!]_{GSB}(s_G)$ in the $\mathcal{GSB}$ semantics and $\lambda \sigma_L.[\![\alpha = \beta]\!]_{LSB}(\sigma_L)$ in the $\mathcal{LSB}$ semantics.

B.2 Properties of the $\mathcal{GSB}$ Semantics

In this section, we introduce the notions of *heap paths* and *generalized heap paths*. We also prove some properties of the $\mathcal{GSB}$ semantics which are used in the proof of Theorem 4.5.3.

Definition B.2.1 (Heap path) *A **heap path** $\zeta = \langle l, \delta \rangle \in Loc \times \Delta$ is a pair consisting of a location and a field path. $HeapPath$ denotes the set $Loc \times \Delta$.*

Definition B.2.2 (Generalized heap path) *A **generalized heap path** $\zeta \in AccPath_p \cup HeapPath$ of a procedure p is an access path of p or a heap path. $GHeapPath_p$ denotes the set of all generalized heap paths of procedure p. $GHeapPath$ denotes the union of all generalized heap paths of all procedures in a program.*

Definition B.2.3 (Generalized heap path value) *The value of a generalized heap path ζ in memory state $\langle L, \rho, h \rangle$ of procedure p is defined to be:*

$$[\![\zeta]\!]_G \langle L, \rho, h \rangle = \begin{cases} \hat{h}(\rho(x), \delta) & \zeta = \langle x, \delta \rangle,\ x \in V_p \\ \hat{h}(l, \delta) & \zeta = \langle l, \delta \rangle,\ l \in L \end{cases}$$

where $\hat{h}$ is as defined in Definition 2.4.4. Note that the above definition generalizes Definition 2.4.4 (value of an access path in the $\mathcal{GSB}$ semantics). The following definition generalizes Definition 2.4.6 (equality of access paths in the $\mathcal{GSB}$ semantics).

Definition B.2.4 (Generalized heap path equality) *Generalized heap paths ζ_1 and ζ_2 are **equal** in a given state s_G, denoted by $[\![\zeta_1 = \zeta_2]\!]_G(s_G)$, if they have the same value in that state, i.e., $[\![\zeta_1]\!]_G(s_G) = [\![\zeta_2]\!]_G(s_G)$. A generalized heap path ζ is **equal to null** in a given state s_G, denoted by $[\![\zeta = \mathtt{null}]\!]_G(s_G)$, if $[\![\zeta]\!]_G(s_G) = null$.*

The following lemma states that a procedure invocation cannot modify fields of objects that are allocated, but which are not reachable from an actual parameter, when a procedure is invoked.

Lemma B.2.5 (Unreachable locations not modified) *Let s^c_G, s^r_G be states in $\mathcal{S}_G$ such that $\langle y = p(x_1, \ldots, x_k), s^c_G\rangle \overset{GSB}{\rightsquigarrow} s^r_G$. Let $L^{reach} \subseteq L^c$ be the location in L^c that are reachable from the actual arguments at s^c_G, i.e.,*

$$L^{reach} = \bigcup_{1 \leq i \leq k} \{l \in L^c \mid \exists \delta \in \Delta, l = [\![\langle x_i, \delta\rangle]\!]_G(s^c_G)\}.$$

For any generalized access path $\zeta = \langle r, \delta\rangle \in GHeapPath$ such that $[\![\zeta]\!]_G(s^c_G) \in L^c \setminus L^{reach}$ the following holds:

1. $[\![\zeta]\!]_G(s^c_G) = [\![\zeta]\!]_G(s^r_G)$, *and*
2. *for any $f \in \mathcal{F}$, $[\![\langle r, \delta f\rangle]\!]_G(s^c_G) = [\![\langle r, \delta f\rangle]\!]_G(s^r_G)$.*

Sketch of Proof: The lemma states that a procedure cannot modify the content of locations it cannot access (reach). The proof is by induction on the derivation tree. We track the set of reachable locations from every variable of the invoked procedure and prove that a variable cannot point-to (and thus potentially modify) locations that are allocated when the procedure is invoked and are not reachable from any actual parameter. Note that for any $l \in L^{reach}$ and any $\delta \in \Delta$, $[\![\langle l, \delta\rangle]\!]_G(s^r_G)$ is defined because $L^c \subseteq L^r$.

The following lemma formally states that any access path that extends a null valued access path has a null value. Similarly, any prefix of a non null valued access path points to a location.

Lemma B.2.6 (Null valued access paths) *Let $s_G \in \mathcal{S}^q_G$ be a $\mathcal{GSB}$ state for procedure q,*

1. *For a generalized heap path $\alpha = \langle r, \delta_0\delta_1\rangle \in GHeapPath_q$, it holds that $[\![\alpha]\!]_G(s_G) = [\![\langle [\![\langle r, \delta_0\rangle]\!]_G(s_G), \delta_1\rangle]\!]_G(s_G)$.*
2. *For any (generalized) access path $\alpha \in GHeapPath_q$,*
   a. *if $[\![\alpha = \texttt{null}]\!]_{GSB}(s_G)$, then for any generalized heap path α' such that $\alpha \leq \alpha'$, $[\![\alpha' = \texttt{null}]\!]_{GSB}(s_G)$,*
   b. *if $[\![\alpha \neq \texttt{null}]\!]_{GSB}(s_G)$, then for any generalized heap path α' such that $\alpha' \leq \alpha$, $[\![\alpha' \neq \texttt{null}]\!]_{GSB}(s_G)$.*

Proof: Immediate from Definition B.2.3 (generalized heap-path value).

Appendix C

Appendix for Chapter 3

This chapter provides formal details pertaining to Chapter 3:

- Section C.1 describes the meaning of control statements.
- Section C.2 provides formal details of the $\mathcal{LCPF}$ semantics.
- Section C.3 describes the analysis of sorting and list manipulating programs.

C.1 The Meaning of Control Statements

Control statements are handled as descried in Section B.1. The semantics function which maps conditional involving comparison between pointer access paths to predicates over states in the $\mathcal{LSL}^{\text{CPF}}$ semantics is

$$\mathcal{B}_{\Sigma_{L_{CPF}}} : \mathit{cond} \rightarrow \Sigma_{L_{CPF}} \rightarrow \{tt, ff\}$$

$$\mathcal{B}_{\Sigma_{L_{CPF}}}[\![\mathit{cnd}]\!] = \lambda\sigma_{L_{cpf}} . \begin{cases} [\![\alpha = \beta]\!]_L^{cpf}(\sigma_{L_{cpf}}) & \mathit{cnd} \equiv \alpha = \beta \\ \neg[\![\alpha = \beta]\!]_L^{cpf}(\sigma_{L_{cpf}}) & \mathit{cnd} \equiv \alpha \neq \beta \\ [\![\alpha = \texttt{null}]\!]_L^{cpf}(\sigma_{L_{cpf}}) & \mathit{cnd} \equiv \alpha = \texttt{null} \\ \neg[\![\alpha = \texttt{null}]\!]_L^{cpf}(\sigma_{L_{cpf}}) & \mathit{cnd} \equiv alpha \neq\!\!= \texttt{null} \end{cases}$$

C.2 Formal Properties of the $\mathcal{LCPF}$ Semantics

This section provides formal details which were omitted from the body of Section 3.7.2.

- Section C.2.1 provides the operational semantics for the intraprocedural statements and the predicate-update formulae for the instrumentation predicates for interprocedural statements.

C.2.1 Formal Specification of the Operational Semantics

This appendix provides the operational semantics for the intraprocedural statements (Section C.2.1.1) and the predicate-update formulae for the instrumentation predicates for interprocedural statements (Section C.2.1.2).

C.2.1.1 Operational Semantics for Atomic Statements

The operational semantics for assignments is specified by *predicate-update formulae*: for every predicate p and for every statement st, the value of p in the 2-valued structure which results by applying st to S, is defined in terms of a formula evaluated over S.

Predicate-update formulae of the core-predicates for assignments are given in Figure C.1. The table also specifies the side condition which enables that application of the statement. These conditions check that null-dereference is not performed. The value of every core-predicate p after the statement executes, denoted by p', is defined in terms of the core predicate values before the statement executes (denoted without primes). Core predicates whose update formula is not specified, are assumed to be unchanged, i.e., $p'(v_1, \ldots) = p(v_1, \ldots)$.

Statement	Predicate-update formulae	side-condition
$\texttt{y} = \texttt{null}$	$y'(v) = 0$	
$\texttt{y} = \texttt{x}$	$y'(v) = x(v)$	
$\texttt{y} = \texttt{x.f}$	$y'(v) = \exists v_1 \colon x(v_1) \wedge f(v_1, v)$	$\exists v_1 \colon x(v_1)$
$\texttt{y.f} = \texttt{null}$	$f'(v_1, v_2) = f(v_1, v_2) \wedge \neg y(v_1)$	$\exists v_1 \colon y(v_1)$
$\texttt{y.f} = \texttt{x}$	$f'(v_1, v_2) = f(v_1, v_2) \vee (y(v_1) \wedge x(v_2))$	$\exists v_1 \colon y(v_1)$
$\texttt{y} = \texttt{alloc(T)}$	$T'(v) = T'(v) \vee new(v)$	
	$eq'(v_1, v_2) = eq(v_1, v_2) \vee new(v_1) \wedge new(v_2)$	
	$new'(v) = 0$	

Figure C.1: Predicate-update formulae defining the operational semantics of assignments.

None of the assignments, except for object allocation, modifies the underlying universe. Object allocation is handled as in [SRW02]: A new individual is added to the universe to represent the allocated object; the auxiliary predicate new is set to hold *only* at that individual; only then, the predicate-update formulae is evaluated.

C.2.1.2 Predicate Update Formulae for Instrumentation Predicates

Figure C.2 provides the update formulae for instrumentation predicates used by the procedure call rule. We use $PT_X(v)$ as a shorthand for $\bigvee_{x \in X} x(v)$. The intended meaning of this formula is to specify that v is pointed to by some variable from $X \subseteq \mathit{Local}^\star$. We use $\mathit{bypass}_X(v_1, v_2)$ as a shorthand for $(F(v_1, v_2) \wedge \neg PT_X(v_1))^*$. The intended meaning of this formula is to specify that v_2 is reachable from v_1 by a path that does not traverse any object which is pointed-to by any variable in $X \subseteq \mathit{Local}^\star$. As we can see, formula $match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_1, v_2)$ again plays a central role.

a. Predicate update formulae for $updCall_q^{y=p(x_1,\ldots,x_k)}$

$$ils'(v) = ils(v) \wedge (\neg PT_{x_1,\ldots,x_k}(v) \vee \exists v_1, v_2 \colon R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_1, v) \wedge F(v_2, v) \wedge \neg eq(v_1, v_2))$$

$$r'_y(v) = \begin{cases} r_{x_i}(v) & : \quad y = h_i \\ 0 & : \quad y \in Local^\star \setminus \{h_1, \ldots, h_k\} \end{cases}$$

b. Predicate update formulae for $updRet_q^{y=p(x_1,\ldots,x_k)}$

$$\begin{aligned}
ils'(v) &= ils(v) \wedge (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v)) \vee \\
&\quad PT_{x_1,\ldots,x_k}(v) \wedge \exists v_1, v_2, v_3 \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) \wedge \neg eq(v_2, v_3) \wedge \\
&\qquad inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_2, v_1) \wedge \\
&\qquad\quad (inUc(v_3) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_3) \wedge F(v_3, v_1) \vee inUx(v_3) \wedge F(v_3, v)) \\
r'_{obj}(v_1, v_2) &= r_{obj}(v_1, v_2) \wedge inUx(v_1) \wedge inUx(v_2) \vee \\
&\quad r_{obj}(v_1, v_2) \wedge inUc(v_1) \wedge inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee \\
&\qquad inUc(v_1) \wedge inUx(v_2) \wedge \exists v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge \\
&\qquad\quad bypass_{\{x_1,\ldots,x_k\}}(v_1, v_a) \wedge r_{obj}(v_f, v_2) \\
r'_x(v) &= inUc(v) \wedge r_x(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee \\
&\qquad inUx(v) \wedge \exists v_x, v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge \\
&\qquad\quad x(v_x) \wedge bypass_{\{x_1,\ldots,x_k\}}(v_x, v_a) \wedge r_{obj}(v_f, v)
\end{aligned}$$

Figure C.2: Predicate-update formulae for the instrumentation predicates used in the procedure call rule. We give the semantics for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q. We assume that p's formal parameters are $h_1, \ldots, h_k$.

Core Predicates	
predicate	**Intended Meaning**
$dle(v_1, v_2)$	The `data` component of v_1 is less-than-or-equal-to the data component of v_2

Instrumentation Predicates		
predicate	**Defining Formula**	**Intended Meaning**
$O(v)$	$\forall v_1 : n(v, v_1) \implies dle(v, v_1)$	v's data field is not strictly greater than that of its successor

Figure C.3: Core and instrumentation predicates used in the analysis of sorting programs.

C.3 Analyzing Sorting Programs

The analysis presented in [LARSW00] allows to prove partial correctness of sorting and list manipulating procedures. In this section, we briefly describe our adaptation of their abstraction to local-heaps.

The main idea is to track the relative order between the data components of list elements using a binary core relation (dle). In addition, an additional instrumentation predicate records the relative order between the data components of successive list elements (O). Figure C.3 lists the additional predicates we use here, together with their intended meaning.

Concrete memory states are represented using *2-valued* logical structures and we abstract them into *3-valued* logical structures using *canonical abstraction*. However, we represent two individuals that differ only in the value of the predicate O by a single summary individual. This improves the performance of the analysis, but may reduce its precision.

We use the same predicate-update formulae as in [LARSW00] to specify the effect of intraprocedural statements on the values of the added predicates. The main idea is to assume (and verify) that a list element is allocated with a random data field, which is not modified afterwards. The program can establish the relative relation between the data fields using comparisons.

The only change in the procedure call rule amounts to updating the rule to construct the memory state at the return site. Note that the values of both the O-predicate and the dle-predicate at the entry to the callee is the same as it was at the call-site.

At the return site, the value of the O-predicate can be taken as is from the call site (for objects not passed to the invoked procedure) and from the exit site (for objects in the invoked procedure local-heap). This is possible because the invoked procedure could not have modified the external references into its local-heap. In addition, if it also has not changed the data-value, the O-predicate still holds for those objects whose successor was passed to the callee.

The dle relation between objects that were (resp. were not) passed to the callee does not change. The only difficulty is in restoring the dle relation between objects that were not passed to the invoked procedure and those that were. The main problem is that we cannot relate an individual that represents a certain heap allocated object at the call site, with the individual that represents the same object at the exit site. This information is lost in our *semantics* for all objects, except for the ones that are pointed to by parameters. Thus, we try to restore the relative relation using the parameters, resorting to an indefinite value as our last choice. The predicate update formulae for the dle-predicate is given in Figure C.4.

To verify that `reverse` returns a list in reversed order, we needed an additional instrumentation predicate $RO(v)$, whose defining formula is $\forall v_1 : n(v, v_1) \implies dle(v_1, v)$. This predicate holds for an individual v, if v's data field is not strictly less than that of its successor [LARSW00].

C.3.1 Verifying the Partial Correctness of Quicksort.

We now describe the analysis of `quicksort` in more details. Figure C.5 shows a program that allocates a random list and sorts it using the `quicksort` procedure. The quicksort procedure partitions the list by moving all the list elements whose data field is less than that of the pivot (the first parameter) to the beginning of the list. The list is then divided into two sub lists, which are sorted in a recursive fashion. Finally, the two lists are linked together. Note that the pivot is strictly larger than all other elements in the first list. This ensures that when the first recursive call returns, the pivot remains the last element in the first sublist.

$$
\begin{array}{l}
dle'(v_1, v_2) = dle(v_1, v_2) \wedge (inUc(v_1) \wedge inUc(v_2) \vee inUx(v_1) \wedge inUx(v_2)) \vee \\
\quad inUc(v_1) \wedge inUx(v_2) \wedge \\
\qquad \left(\left\{ \begin{array}{lcl} 1 & : & \exists v_c, v_x \colon match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \wedge dle(v_1, v_c) \wedge dle(v_x, v_2) \\ 0 & : & \exists v_c, v_x \colon match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \wedge \neg dle(v_1, v_c) \wedge \neg dle(v_x, v_2) \\ \frac{1}{2} & : & \text{otherwise} \end{array} \right. \right) \vee \\
\quad inUx(v_1) \wedge inUc(v_2) \wedge \\
\qquad \left(\left\{ \begin{array}{lcl} 1 & : & \exists v_c, v_x \colon match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \wedge dle(v_1, v_c) \wedge dle(v_x, v_2) \\ 0 & : & \exists v_c, v_x \colon match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \wedge \neg dle(v_1, v_c) \wedge \neg dle(v_x, v_2) \\ \frac{1}{2} & : & \text{otherwise} \end{array} \right. \right)
\end{array}
$$

Figure C.4: The predicate update formulae for the dle predicate used in the procedure call rule to construct the memory state at the return site.

We made two modifications to the program in order to eliminate two (false, i.e., dead) cutpoints. The n-field of the pivot object (pointed-to by `p`) is dead when the first recursive call occurs. Had we not nullified it, the objects pointed-to by `tl` and `last` would have become cutpoints. Similarly, the `pr` is dead when the second recursive call occurs. However, the object it points to would have become a cutpoint had it not been nullified. The nullification of all the local variables prior to the return statement is only conducted to simplify the presentation.

Figure C.6 shows several concrete states that occur during the execution of `quicksort` and their abstraction. For clarity, in the concrete states we do not draw the instrumentation predicates. Instead, we draw in each object the numeric value of its data field, from which the dle relation can be easily inferred. In the abstract states, we do not draw the dle and r_{obj} relations. We prove that a list is sorted by showing that the predicate O holds in all its elements. Because this predicate does not take a role in the abstraction and we only care when its value is 1, we draw it only in those nodes in which it hold. If we do not draw it in a node u, we mean that its value in u is either 0 or $\frac{1}{2}$. Because the dle is a transitive relation, we do not draw dle-labeled edges which can be inferred. For example, if the data field of u_1 is less-or-equal to the data field of u_2, and the data field of the latter is less-or-equal to the data field of u_3, we draw a dle-labeled edge from u_1 to u_2 and from u_2 to u_3, but we do not draw such an edge from u_1 to u_3. Because the dle is reflexive, we omit self dle-loops in non summary objects.

Note that the second recursive call is invoked on a local-heap which, under abstraction, is identical to the local-heap that was passed by the external call, thus the results of analyzing the recursive call can be reused in the analysis of the external call. Also note that when the procedure returns, every element has the O property, thus, proving that the list is in order.

```
public static List quickSortRec(List p, List q) {
 // Location (A)
 List hd,pr,tl;
 if (p == null)
  return first;

 if (p == q)
  return h2;

 hd = p;
 pr = p;
 tl = p.n;

 // Partitioning of the list
 while (tl != q) {
  if (tl.d < p.d) {
   pr.n = tl.n;
   tl.n = hd;
   hd = tl;
   tl = pr.n;
  }
  else {
   pr = tl;
   tl = tl.n;
  }
 }

 tl = p.n;
 p.n = null; // removing a false cutpoint due to a dead field
 pr = null; // removing a false cutpoint due to a dead variable

 // Location (B)
 List s = quickSortRec(hd,p);
 // Location (C)
 List t = quickSortRec(tl,q);
 // Location (D)

 p.n = t;

 t = hd = pr = tl = null;

 // Location (E)
 return s;
}

public static void main(String argv[]) {
 List x = randomList(8);
 List y = quicksort(x,null);
}
```

Figure C.5: A program that sorts a list using a quicksort algorithm for singly linked lists.

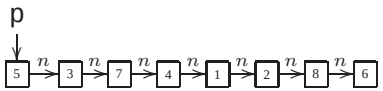
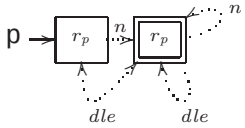
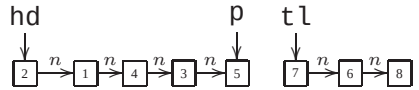
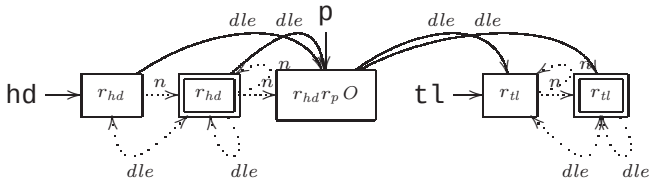
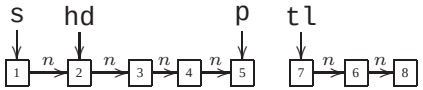
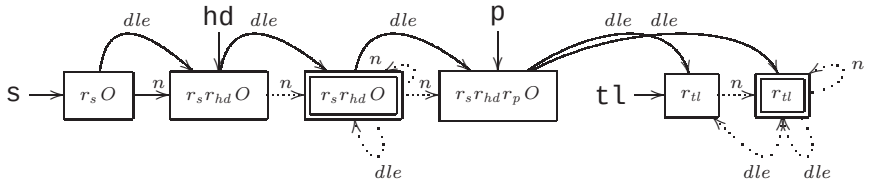
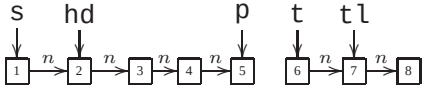
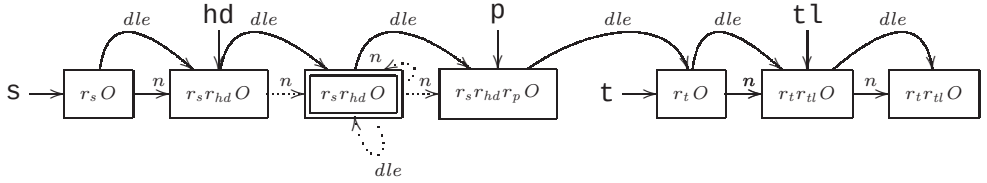
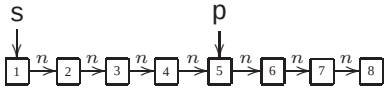
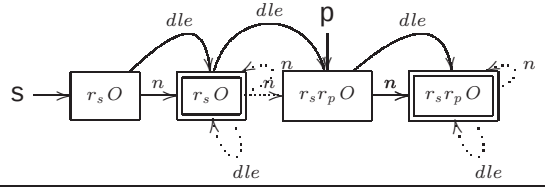
Location	Concrete Memory States	Abstract Memory States
(A)		
(B)		
(C)		
(D)		
(E)		

Figure C.6: Concrete memory states that occur during the execution of `quicksort` and their abstractions.

Appendix D

Appendix for Chapter 4

This chapter provides formal details pertaining to Chapter 4:

- Section D.1 describes the meaning of control statements.
- Section D.2 provides the proofs of the main theorems stated in Section 4.5.
- Section D.3 provides additional formal details pertaining to the concrete semantics presented in Section 4.8.2.
- Section D.4 shows that Deutsch's abstract-interpretation algorithm [Deu94] can be seen as an abstraction of the $\mathcal{LSL}$ semantics and provides insight into the clever interprocedural aspects of the analysis.

D.1 The Meaning of Control Statements

Control statements are handled as descried in Section B.1. The semantics function which maps conditional involving comparison between pointer access paths to predicates over states in the $\mathcal{LSL}$ semantics is

$$\mathcal{B}_{\Sigma_L} : \mathit{cond} \to \Sigma_L \to \{tt, ff\}$$

$$\mathcal{B}_{\Sigma_L}[\![\mathit{cnd}]\!] = \lambda.\begin{cases} [\![\alpha = \beta]\!]_{LSL}(\sigma_L) & \mathit{cnd} \equiv \alpha = \beta \\ \neg[\![\alpha = \beta]\!]_{LSL}(\sigma_L) & \mathit{cnd} \equiv \alpha \neq \beta \\ [\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L) & \mathit{cnd} \equiv \alpha = \texttt{null} \\ \neg[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L) & \mathit{cnd} \equiv alpha \neq\!\neq \texttt{null} \end{cases}$$

D.2 Formal Properties of the $\mathcal{LSL}$ Semantics

This section provides the proofs of the main theorems stated in Section 4.5. Furthermore, it proves a stronger theorem than the equivalence theorem, i.e., the preservation of *context-aware equivalence*.

More specifically, in this section we prove our main theorem, Theorem 4.5.3 (preservation of observational equivalence). In Section D.2.1, we state, and prove, additional properties of the $\mathcal{LSL}$ semantics. In Section D.2.2, we define the notion of context-aware equivalence between states in $\mathcal{LSL}$ and states in $\mathcal{GSB}$, and prove a stronger theorem than the equivalence theorem, i.e., the preservation of *context-aware equivalence*.

As in Section 4.5, we assume, A and *CPL* with a certain index (resp. prime) to be the heap, resp. cutpoint-labels component of a state σ_L with the same index (resp. prime). Similarly, we assume L, ρ, and h with a certain index (resp. prime) to be the set of allocated locations, resp. environment, resp. heap of a state s_G with the same index (resp. prime). In addition, we use the notations $[\![\alpha \neq \beta]\!]_{LSL}(\sigma_L)$, $[\![\alpha \neq \texttt{null}]\!]_{LSL}(\sigma_L)$, $[\![\alpha \neq \beta]\!]_{GSB}(s_G)$, and $[\![\alpha \neq \texttt{null}]\!]_{GSB}(s_G)$ as shorthand for $\neg[\![\alpha = \beta]\!]_{LSL}(\sigma_L)$, resp. $\neg[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L)$, resp. $\neg[\![\alpha = \beta]\!]_{GSB}(s_G)$, resp. $\neg[\![\alpha = \texttt{null}]\!]_{GSB}(s_G)$.

D.2.1 Properties of the $\mathcal{LSL}$ Semantics

In this section, we prove certain properties of the $\mathcal{LSL}$ semantics. These properties are needed in the proof of the Context-Aware Equivalence theorem, however, they are mere technicalities of the definition of $\mathcal{LSL}$ as given

in Figure 4.6 and Figure 4.7.

The following lemma establishes some of the properties of the $[\cdot]_{\cdot}$ function defined in Figure 4.5. In particular, it states certain properties related to equality of access-paths.

Lemma D.2.1 (Properties of $[\cdot]_{\cdot}$) *Let $\sigma_L = \langle \text{CPL}, A \rangle \in \Sigma_L^q$ be an (admissible) memory state for procedure q. For any generalized access paths $\alpha, \beta \in GAccPath_q$ the following holds:*

1. $[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L) \iff [\alpha]_A = \emptyset$
2. $[\![\alpha = \beta]\!]_{LSL}(\sigma_L) \iff [\alpha]_A = [\beta]_A$
3. $[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L) \implies (\forall \alpha', \alpha \leq \alpha' \implies [\![\alpha' = \texttt{null}]\!]_{LSL}(\sigma_L))$
4. $[\![\alpha \neq \texttt{null}]\!]_{LSL}(\sigma_L) \implies (\forall \alpha', \alpha' \leq \alpha \implies [\![\alpha' \neq \texttt{null}]\!]_{LSL}(\sigma_L))$
5. $[\alpha]_A \in A \cup \{\emptyset\}$
6. $\forall cpl \in CPL : [\langle cpl, \epsilon \rangle]_A \neq \emptyset$

Sketch of Proof: 1-5 are immediate from the definitions of: $[\![\cdot = \cdot]\!]_{LSL}$, $[\cdot]_{\cdot}$, and admissibility. Lemma D.2.1(6) is proven using an induction on the derivation tree where the key observation is that objects never lose their labels, i.e., an access path of the form $\langle cpl, \epsilon \rangle$, where $cpl \in CPL$, is never removed from the description of an object.

The following lemma states certain properties of the sets of objects and the various mappings defined in the procedure call inference rule (see Figure 4.7).

Lemma D.2.2 (Properties of the procedure call inference rule) *Let $\sigma_L^c = \langle \text{CPL}^c, A^c \rangle \in \Sigma_L^q$ be an admissible memory state for procedure q in which the statement $y = p(x_1, \ldots, x_k)$ is executed. Let $\langle \text{CPL}^e, A^e \rangle$, $\langle \text{CPL}^x, A^x \rangle$, $\langle \text{CPL}^r, A^r \rangle$, O_c^{args}, O_c^{passed}, O_c^{cp}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, $bind_{call}$, and $bind_{ret}$ be as defined in Figure 4.7. Let $\alpha \in GAccPath_q$ be an arbitrary generalized access path of procedure q. The following holds.*

1. $\emptyset \notin O_c^{passed}$.
2. $(A^c \setminus O_c^{passed}) \cap map(sub(bind_{ret}))\ A^x = \emptyset$.
3. *If* $[\alpha]_{A^r} \neq \emptyset$ *and* $[\alpha]_{A^r} \notin (A^c \setminus O_c^{passed})$, *then* $[\alpha]_{A^r} \in map(sub(bind_{ret}))\ A^x$.
4. $\forall a, a' \in dom(bind_{ret}) : a \neq a' \implies \forall \alpha \in bind_{ret}(a) : \forall \beta \in bind_{ret}(a') : \alpha \not\leq \beta \wedge \beta \not\leq \alpha$.
5. $\forall a \in dom(bind_{ret}) : \forall \alpha, \beta \in bind_{ret}(a) : \alpha \neq \beta \implies \alpha \not\leq \beta \wedge \beta \not\leq \alpha$.
6. $\forall a \in range(bind_{ret}) : \forall \alpha \in a . \forall \alpha' < \alpha, [\alpha']_{A^c} \notin O_c^{passed}$.
7. $\emptyset \notin range(bind_{ret})$

Proof: We only sketch the proof of 3. The other properties are derived immediately from the definition of the $\mathcal{LSL}$ semantics.

1.	$[\alpha]_{A^r} \neq \emptyset$, $[\alpha]_{A^r} \notin A^c \setminus O_c^{passed}$	Assumption
2.	$A^r = (A^c \setminus O_c^{passed}) \cup map(sub(bind_{ret}))\ A^x$	See $Figure$ 4.7.
3.	$(A^c \setminus O_c^{passed}) \cap map(sub(bind_{ret}))\ A^x = \emptyset$	$Lemma\ D.2.2(2)$
4.	$[\alpha]_{A^r} \in map(sub(bind_{ret}))\ A^x$	$1-3$

The following lemma establishes certain properties of the memory states that occur during a procedure invocation at the call-site, at the entry-site, at the exit-site, and at the return-site. Informally, it states the following properties:

1. Properties of cutpoint-labels:

   (a) Cutpoint-labels are never empty.

   (b) At the entry state, every access path in a cutpoint-label points to the corresponding cutpoint.

2. When a procedure returns, an access path can point to one of the following: to an object which was not passed to the procedure, to an object that was in the invoked procedure local-heap, or to null.

3. When a procedure returns, an access path that points to an object which was not in the callee's local-heap does not point to such an object when the procedure has been invoked.

4. The function $sub(bind_{ret})$ is injective for all objects that are reachable at the return site. Furthermore, it maps all unreachable objects to the empty set.

5. When a procedure returns, every access path that points to the invoked procedure's local-heap, has a unique prefix which starts either with the return value, an object pointed-to by an actual parameter, or a cutpoint of that invocation.

Lemma D.2.3 (Properties of procedure calls) *Let* σ_L^c, q, $y = p(x_1, \ldots, x_k)$, $\langle \text{CPL}^e, A^e \rangle$, $\langle \text{CPL}^x, A^x \rangle$, $\langle \text{CPL}^r, A^r \rangle$, O_c^{args}, O_c^{passed}, O_c^{cp}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, $bind_{call}$, $bind_{ret}$, *and* $\alpha \in GAccPath_q$ *be as in Lemma D.2.2. The following holds:*

1. *For any* $cpl \in CPL^e$*, the following holds:*

   (a) $\emptyset \neq cpl \subseteq F_q \times \Delta$*, and*

   (b) for any $\alpha' \in cpl$ *and any* $\delta \in \Delta$, $[\![\alpha'.\delta = \langle cpl, \delta \rangle]\!]_{LSL}(\sigma_L^e)$.

2. *If* $[\alpha]_{A^r} \notin (A^c \setminus O_c^{passed})$*, then for any generalized access path* α' *such that* $\alpha \leq \alpha'$, $[\alpha']_{A^r} \in \{\emptyset\} \cup map(sub(bind_{ret}))\ A^x$.

3. *If* $[\alpha]_{A^r} \in (A^c \setminus O_c^{passed})$*, then for every generalized access path* $\alpha' \leq \alpha$ *it holds that* $[\alpha']_{A^c} \notin O_c^{passed}$.

4. *For any* $o, o' \in A^x$*, if* $o \neq o'$*, then either* $sub(bind_{ret})\ o \neq sub(bind_{ret})\ o'$ *or* $sub(bind_{ret})\ o = sub(bind_{ret})\ o' = \emptyset$.

5. *(a) If* $\langle y, \epsilon \rangle \leq \alpha$*, then* $[\![\langle y, \delta \rangle = \texttt{null}]\!]_{LSL}(\sigma_L^r) \iff [\![\langle ret, \delta \rangle = \texttt{null}]\!]_{LSL}(\sigma_L^x)$

   (b) If $\langle y, \epsilon \rangle \not\leq \alpha$ *and* $\forall \alpha' \leq \alpha.[\alpha']_{A^c} \notin O_c^{passed}$*, then* $[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L^r)) \iff [\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L^c)$.

   (c) If $[\alpha]_{A^c} \in O_c^{args} \cup O_c^{cp}$ *and* $\alpha \in Bypass(O_c^{passed})\ [\alpha]_{A^c}$ *then there exists a (unique)* $r_1^\alpha \in dom(bind_{ret})$ *such that* $\alpha \in bind_{ret}\ r_1^\alpha$*. Furthermore for any* $\delta \in \Delta$, $[\![\alpha.\delta = \texttt{null}]\!]_{LSL}(\sigma_L^r) \iff \forall \alpha_p \in r_1^\alpha, [\![\alpha_p.\delta = \texttt{null}]\!]_{LSL}(\sigma_L^x)$

6. *For any* $o \in A^x$ *such that* $[\alpha]_{A^r} = sub(bind_{ret})\ o$ *and* $[\alpha]_{A^r} \neq \emptyset$*, there exists a unique* $\alpha_0 \leq \alpha$ *such that* $\alpha_0 \in$ *flat* $(range(bind_{ret}))$*. Furthermore, one (and only one) of the following holds:*

   (a) $\alpha_0 = \langle y, \epsilon \rangle$

   (b) $[\alpha_0]_{A^c} \in O_c^{args}$ *and* $\alpha_0 \in Bypass(O_c^{passed})\ [\alpha_0]_{A^c}$

   (c) $[\alpha_0]_{A^c} \in O_c^{cp}$ *and* $\alpha_0 \in Bypass(O_c^{passed})\ [\alpha_0]_{A^c}$

   and, in addition, $r_1^\alpha.\delta_1^\alpha \subseteq o$ *where* $\alpha = \alpha_0.\delta_1^\alpha$, $r_1^\alpha.\delta_1^\alpha \neq \emptyset$ *and*

$$r_1^\alpha = \begin{cases} \{\langle ret, \epsilon \rangle\} & (\textit{if case } 6a \textit{ holds}) \\ bind_{args}\ [\alpha_0]_{A^c} & (\textit{if case } 6b \textit{ holds}) \\ bind_{cp}\ [\alpha_0]_{A^c} & (\textit{if case } 6c \textit{ holds}) \end{cases}$$

Proof:

Properties 2–5 are immediate. We prove properties 1 and 6.

1.

(i) By definition, $CPL^e = map(sub(bind_{args}))\ O_c^{cp}$. To show that $\emptyset \notin CPL^e \subseteq 2^{F_q \times \Delta}$, we show that $\forall o \in O_c^{passed}.\emptyset \neq sub(bind_{args})\ o \subseteq F_q \times \Delta$. This proves (i), because $O_c^{cp} \subseteq O_c^{passed}$. Recall that

$$O_c^{passed} \quad = \quad \mathit{RObjs}(A^c)O_c^{args} = \{o \in A^c \mid o' \in O_c^{args}, \delta \in \Delta, o'.\delta \subseteq o\}$$

Thus,

5. $o \in O_c^{passed} \iff \exists o' \in O_c^{args}, \exists \delta \in \Delta.\ o'.\delta \subseteq o.$

By definition (see Figure 4.7),

6. $\emptyset \notin O_c^{args}$, and
7. $bind_{args} = \lambda o \in O_c^{args}.\{\langle h_i, \epsilon\rangle \mid 1 \leq i \leq k,\ x_i \in o\}$.

Thus, for any $o \in O_c^{passed}$,

8. $\begin{array}{lcl} sub(bind_{args})(o) & = & \mathit{flat}\,\{bind_{args}(a).\delta \mid a \in dom(bind_{args}), \delta \in \Delta, a.\delta \subseteq o\} \\ & = & \mathit{flat}\,\{bind_{args}(o').\delta \mid o' \in O_c^{args}, \delta \in \Delta, o'.\delta \subseteq o\} \end{array}$

which gives (by $5, 6$) that $\emptyset \notin sub(bind_{args})(o)$, and (by $5, 7$) that $sub(bind_{args})(o) \subseteq F_q \times \Delta$.

To prove (ii), we recall (see Figure 4.7) that

9. $CPL^e = map(sub(bind_{args}))\ O_c^{cp}$,
10. $O_c^{cp} \subseteq O_c^{passed}$,
11. $bind_{cp} = \lambda o \in O_c^{cp}.\{\langle sub(bind_{args})\ o, \epsilon\rangle\}$, and
12. $bind_{call} = \lambda o \in O_c^{args} \cup O_c^{cp}.\ \begin{cases} bind_{args}(o) & o \in O_c^{args} \\ bind_{cp}(o) & o \in O_c^{cp} \end{cases}$,
13. $A^e = map(sub(bind_{call}))\ O_c^{passed}$.

Thus, for any $cpl \in CPL^e$ and for any $o \in A^e$,

14. $\langle cpl, \delta\rangle \in o \iff$ 13
15. $\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge\ \langle cpl, \delta\rangle \in sub(bind_{call})o' \iff$
16. $\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge\ \langle cpl, \delta\rangle \in \mathit{flat}\left\{ bind_{call}(a_1).\delta_1 \;\middle|\; \begin{array}{l} a_1 \in dom(bind_{call}), \\ \delta_1 \in \Delta,\ a_1.\delta_1 \subseteq o' \end{array} \right\} \iff$ Def. of sub
17. $\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge\ \exists o'' \in O_c^{cp} : \langle cpl, \delta\rangle = \langle sub(bind_{args})\ o'', \delta\rangle \ \wedge\ o''.\delta \subseteq o' \iff$ $9, 7, 11, 12$
18. $\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge\ \exists o'' \in O_c^{cp} : \langle cpl, \delta\rangle = \langle sub(bind_{args})\ o'', \delta\rangle \ \wedge\ \forall o''' \in O_c^{args} : \forall \delta' \in \Delta : o'''.\delta' \in o'' \implies o'''.\delta'.\delta \in o' \iff$ Admissibility of σ_L^c, $O_c^{passed} = \mathit{RObjs}(A^c)O_c^{args}$
19. $\exists o' \in O_c^{passed} : \exists o'' \in O_c^{cp} : \langle cpl, \delta\rangle = \langle sub(bind_{args})\ o'', \delta\rangle \ \wedge\ sub(bind_{args})\ o''.\delta \subseteq sub(bind_{args})\ o' \iff$ 18, 8, Def. of $bind_{args}$
20. $\forall \alpha' \in cpl : \alpha'.\delta \in o \iff$ 19

6.

21.	$[\alpha]_{A^r} \neq \emptyset, o \in A^x, [\alpha]_{A^r} = sub(bind_{ret})\ o$	Assumptions
22.	$\alpha \in sub(bind_{ret})\ o$	21, Def. of $[\cdot]$.
23.	$\alpha \in \mathit{flat}\,\{bind_{ret}(a).\delta_1^\alpha \mid a \in dom(bind_{ret}),\ \delta_1^\alpha \in \Delta, a.\delta_1^\alpha \subseteq o\}$	22, Def. of sub
24.	$\exists a \in dom(bind_{ret})\colon \exists \delta_1^\alpha \in \Delta\colon a.\delta_1^\alpha \subseteq o \wedge \alpha \in (bind_{ret}\ a).\delta_1^\alpha$	23, Def. of *flat*
25.	$\exists a \in dom(bind_{ret})\colon \exists \delta_1^\alpha \in \Delta\colon \exists \alpha_0 \in bind_{ret}(a)\colon a.\delta_1^\alpha \subseteq o \wedge \alpha = \alpha_0.\delta_1^\alpha$	24, Def. of $\cdot.\cdot$
26.	$\exists! a \in dom(bind_{ret})\colon \exists \delta_1^\alpha \in \Delta\colon \exists \alpha_0 \in bind_{ret}(a)\colon a.\delta_1^\alpha \subseteq o \wedge \alpha = \alpha_0.\delta_1^\alpha$	25, $Lemma\ D.2.2(4)$, $\alpha_0 \leq \alpha$
27.	$\exists! a \in dom(bind_{ret})\colon \exists! \delta_1^\alpha \in \Delta\colon \exists! \alpha_0 \in bind_{ret}(a)\colon a.\delta_1^\alpha \subseteq o \wedge \alpha = \alpha_0.\delta_1^\alpha$	26, $Lemma\ D.2.2(5)$, $\alpha_0 \leq \alpha$
28.	Let $a, \alpha_0, \delta_1^\alpha$ be the unique values satisfying 27. We continue with a case analysis of the possible values of $a \in dom(bind_{ret})$	
29.	$\alpha_0 \in \mathit{flat}\,(range(bind_{ret})) \wedge \forall \alpha' \leq \alpha\colon \alpha' \in \mathit{flat}\,(range(bind_{ret})) \implies \alpha' = \alpha_0$	$27 - 28$
30.	$dom(bind_{ret}) = \{\{\langle ret, \epsilon\rangle\}\} \cup map(bind_{args})\ O_c^{args} \cup map(bind_{cp})\ O_c^{cp}$	Def. of $bind_{ret}$
31.	Assume $a = \{\langle ret, \epsilon\rangle\}$ (Case 6a)	
32.	$bind_{ret}(\{\langle ret, \epsilon\rangle\}) = \{\langle y, \epsilon\rangle\}$	Def. of $bind_{ret}$
33.	$\alpha_0 = \langle ret, \epsilon\rangle$	29, 32
34.	$\{\langle ret, \epsilon\rangle\}.\delta_2 \subseteq o$	27, 28, 33
35.	Assume $a \in map(bind_{args})\ O_c^{args}$ (Case 6b)	
36.	$\exists o' \in O_c^{args}, a = bind_{args}\ o'$	35
37.	$\alpha_0 \in bind_{ret}(a)$	28
38.	$\exists o' \in O_c^{args}, a = bind_{args}\ o',\ \alpha_0 \in Bypass(O_c^{passed})\ o'$	$35 - 37$, Def. of $bind_{ret}$
39.	$a = bind_{args}([\alpha_0]_{A^c}),\ \alpha_0 \in Bypass(O_c^{passed})\ [\alpha_0]_{A^c}$	38, Def. of $bind_{ret}$ and $Bypass$
40.	$bind_{args}\ [\alpha_0]_{A^c}.\delta_1^\alpha \subseteq o$	28, 39
41.	Assume $a \in map(bind_{args})\ O_c^{args}$ (Case 6c)	
42.	proof analogous to case 6b	

Note that, by Lemma D.2.2(7), $r_\alpha^1 \neq \emptyset$.

In the following, we sketch the proofs of additional properties of $\mathcal{LSL}$ which are stated in Section 4.5.

Sketch of Proof (Theorem 4.5.9):

(i) For access paths that in memory state σ_L^c point to an object which is not in O_c^{passed}, the proof is immediate from admissibility of σ_L^r and the fact that $A^c \setminus O_c^{passed} \subseteq A^r$.

For accesses paths α, β that in memory state σ_L^c point-to (the same) object in O_c^{passed}, but do not *pass through* any object in O_c^{passed}, the proof follows from Lemma D.2.3(5c).

(ii) For access paths that are equal *null* in σ_L^c, the proof is immediate from Lemma D.2.3(5b).

Sketch of Proof (Theorem 4.5.10):

The proof is done by induction on the shape of the derivation tree. The base case is immediate because in every statement in both σ_L^1 and σ_L^2:

- the same set of access paths that start with a variable, are added / removed from the description of every object, and
- the side-conditions for executing a statement involve only access paths that start with a variable.

The induction step for (non-atomic) intraprocedural statements is also immediate because of the aforementioned nature of side-conditions in the $\mathcal{LSL}$ semantics. To see why the induction step holds for a procedure call, we observe that in both σ_L^1 and σ_L^2:

- the same objects are reachable from the actual parameters, and
- at procedure return, the update of access paths that start with a variable, is done using the same cutpoints.

D.2.2 Context-Aware Equivalence

In this section, we state and prove the context-aware equivalence theorem (Theorem D.2.9). Theorem 4.5.3 is an immediate corollary of Theorem D.2.9.

Definition D.2.4 (Renaming function) *Given an $\mathcal{LSL}$ state $\langle \mathrm{CPL}, A\rangle$ of procedure p, and a $\mathcal{GSB}$ state $\langle L, \rho, h\rangle$ of procedure p, a function $f\colon CPL \to L$ is a **renaming function** if it is total and injective. We lift f to $\hat{f}\colon GAccPath_p \to GHeapPath_p$ as follows:*

$$\hat{f}(\langle r, \delta\rangle) = \begin{cases} \langle r, \delta\rangle & : \ r \in V_p \\ \langle f(r), \delta\rangle & : \ \textit{otherwise} \end{cases}$$

Definition D.2.5 (Context-Aware Equivalence) *Let p be a procedure. The **states** $\sigma_L = \langle \mathrm{CPL}, A\rangle \in \Sigma_L^p$ and $s_G = \langle L, \rho, h\rangle \in \mathcal{S}_G^p$ are **context-aware equivalent w.r.t. a renaming function** $f\colon CPL \to L$, denoted by $\sigma_L \propto_f s_G$, if for all $\alpha, \beta, \gamma \in GAccPath_p$,*

1. $[\![\alpha = \beta]\!]_{LSL}(\sigma_L) \iff [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G)$,
2. $[\![\gamma = \mathtt{null}]\!]_{LSL}(\sigma_L) \iff [\![\hat{f}(\gamma) = \mathtt{null}]\!]_{GSB}(s_G)$.

*The states σ_L and s_G are **context-aware equivalent** if there exists a renaming function f s.t. $\sigma_L \propto_f s_G$.*

The following lemma is rather technical. It states that any extension of a renamed access path points to the same location as the renamed extended access path.

Lemma D.2.6 *Let $\sigma_L \in \Sigma_L^q$ and $s_G \in \mathcal{S}_G^q$ be context-aware equivalent states w.r.t a renaming function f. For any $\alpha, \alpha_0 \in GAccPath_q$ and any $\delta \in \Delta$ such that $\alpha = \alpha_0.\delta$, $[\![\hat{f}(\alpha)]\!]_G(s_G) = [\![\hat{f}(\alpha_0).\delta]\!]_G(s_G) = [\![\langle [\![\hat{f}(\alpha_0)]\!]_G(s_G), \delta\rangle]\!]_G(s_G)$.*

Proof: Immediate from the definition of $\hat{f}$, the definition of $\cdot.\cdot$, and Lemma B.2.6(1).

The following lemma shows that context-aware equivalence at the call-site, implies context-aware equivalence at the entry-site. Furthermore, it defines an appropriate renaming function (f_e). Property 1 shows that f_e is indeed a renaming function and Property 2 proves that the entry states are context-aware equivalent with respect to f_e. Properties 3–5 establish certain properties of f_e.

Lemma D.2.7 (Context-aware equivalence of invoked procedures)
Let $\sigma_L^c = \langle \mathrm{CPL}^c, A^c\rangle \in \Sigma_L^q$ and $s_G^c = \langle L^c, \rho^c, h^c\rangle \in \mathcal{S}_G^q$ be context-aware equivalent states w.r.t. a renaming function f, i.e., $\sigma_L^c \propto_f s_G^c$. Let $y = p(x_1, \ldots, x_k)$ be a call to procedure p whose formal parameters are $h_1, \ldots, h_k$. Let $\sigma_L^e = \langle \mathrm{CPL}^e, A^e\rangle$, $\sigma_L^x = \langle \mathrm{CPL}^e, A^x\rangle$, $\sigma_L^r = \langle \mathrm{CPL}^c, A^r\rangle$, O_c^{args}, O_c^{passed}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, and $bind_{call}$ be as defined in Figure 4.7. Let $s_G^e = \langle L_e, \rho_e, h_e\rangle$, $s_G^x = \langle L_x, \rho_x, h_x\rangle$, and $s_G^r = \langle L_r, \rho_r, h_r\rangle$ be as defined in Figure 2.5. Let L^{reach} be as defined in Lemma B.2.5. Let $f_e\colon CPL^e \to L^e$ such that $f_e(cpl) = [\![\alpha]\!]_G(s_G^e)$ where $\alpha \in cpl$. The following holds:

1. *f_e is a renaming function.*
2. *$\sigma_L^e \propto_{f_e} s_G^e$.*
3. *(a) For any $o \in O_c^{args}$, for any $\alpha_q \in Bypass(O_c^{passed})\ o$, for any $\alpha_p \in bind_{args}\ o$, $[\![\hat{f}(\alpha_q)]\!]_G(s_G^c) = [\![\hat{f}_e(\alpha_p)]\!]_G(s_G^e)$.*
   (b) For any $o \in O_c^{cp}$, for any $\alpha_q \in Bypass(O_c^{passed})\ o$, for any $\alpha_p \in bind_{cp}\ o$, $[\![\hat{f}(\alpha_q)]\!]_G(s_G^c) = [\![\hat{f}_e(\alpha_p)]\!]_G(s_G^e)$.

4. *For any* $\alpha \in GAccPath_q$, $[\alpha]_{A^c} \in O_c^{passed} \iff [\![\hat{f}(\alpha)]\!]_G(s_G^c) \in L^{reach}$

5. *For any* $\alpha_p \in (\{h_1, \ldots, h_k\} \cup CPL^e) \times \{\epsilon\}$, $[\![\hat{f}_e(\alpha_p)]\!]_G(s_G^e) = [\![\hat{f}_e(\alpha_p)]\!]_G(s_G^x)$.

Proof: 1.
The function f_e is a total function from CPL^e to L^e. It is well defined because, by construction of CPL^e:

- every $cpl \in CPL^e$ contains at least one generalized access path (see Lemma D.2.2(1)), and
- for every $\alpha_1, \alpha_2 \in cpl$, $[\![\alpha_1 = \alpha_2]\!]_{LSL}(\sigma_L^e)$.

The function f_e is injective because, by construction of CPL^e, for every $cpl_1, cpl_2 \in CPL^e$ such that $cpl_1 \neq cpl_2$ for every $\alpha_1 \in cpl_1$ and $\alpha_2 \in cpl_2$ $[\![\alpha_1 \neq \alpha_2]\!]_{LSL}(\sigma_L^e)$.
2.

1. σ_L^e and s_G^e are observationally equivalent: For any access paths $\langle h_i, \delta\rangle$ and $\langle h_j, \delta'\rangle$, where $1 \leq i, j \leq k$ and $\delta, \delta' \in \Delta$, $[\![\langle h_i, \delta\rangle = \langle h_j, \delta'\rangle]\!]_{LSL}(\sigma_L^e) \iff [\![\langle x_i, \delta\rangle = \langle x_j, \delta'\rangle]\!]_{LSL}(\sigma_L^c) \iff [\![\langle x_i, \delta\rangle = \langle x_j, \delta'\rangle]\!]_{GSB}(s_G^c) \iff [\![\langle h_i, \delta\rangle = \langle h_j, \delta'\rangle]\!]_{GSB}(s_G^e)$. The proof for the preservation of equality with `null` of access paths that start at a formal variable is analogous. All other variables $x \in V_p \setminus F_p$ are equal to `null` at procedure entry by definition.

2. σ_L^e and s_G^e are context-aware equivalent w.r.t f_e. We prove this by case analysis.

   - Assume $\alpha = \langle h_i, \delta\rangle$ and $\beta = \langle h_j, \delta'\rangle$. Then $[\![\alpha = \beta]\!]_{LSL}(\sigma_L^e) \iff [\![\hat{f}_e(\alpha) = \hat{f}_e(\beta)]\!]_{GSB}(s_G^e)$, because σ_L^e and s_G^e are observationally equivalent; and, by Definition D.2.4, $\alpha = \hat{f}_e(\alpha)$ and $\beta = \hat{f}_e(\beta)$.
   - Assume $\alpha = \langle cpl, \delta_\alpha\rangle$ and $\beta = \langle h, \delta_\beta\rangle$ for some $cpl \in CPL^e$, $\delta_\alpha, \delta_\beta \in \Delta$ and $h \in F_p$

$[\![\alpha = \beta]\!]_{LSL}(\sigma_L^e)$	$\iff$	
$\forall \alpha' \in cpl\colon [\![\alpha'.\delta_\alpha = \beta]\!]_{LSL}(\sigma_L^e)$	$\iff$	Lemma D.2.3(1), transitivity of $[\![\cdot = \cdot]\!]_{LSL}$
$\forall \alpha' \in cpl\colon [\![\alpha'.\delta_\alpha = \beta]\!]_{GSB}(\sigma_L^e)$	$\iff$	σ_L^e and s_G^e are observationally equivalent
$\forall \alpha' \in cpl\colon [\![\hat{f}_e(\alpha'.\delta_\alpha) = \hat{f}_e(\beta)]\!]_{GSB}(s_G^e)$	$\iff$	Definition D.2.4
$\forall \alpha' \in cpl\colon [\![\hat{f}_e(\alpha'.\delta_\alpha)]\!]_G(s_G^e) = [\![\hat{f}_e(\beta)]\!]_G(\sigma_L^e)$	$\iff$	Def. of equality
$\forall \alpha' \in cpl\colon [\![\langle \hat{f}_e(\alpha'), \delta_\alpha\rangle]\!]_G(s_G^e) = [\![\hat{f}_e(\beta)]\!]_G(\sigma_L^e)$	$\iff$	Lemma D.2.6
$[\![\langle f_e(cpl), \delta_\alpha\rangle]\!]_G(s_G^e) = [\![\hat{f}_e(\beta)]\!]_G(\sigma_L^e)$	$\iff$	$f_e(cpl) = [\![\alpha']\!]_G(\sigma_L^e)$, $\alpha' \in cpl,\ cpl \neq \emptyset$
$\forall \alpha' \in cpl\colon [\![\hat{f}_e(\langle cpl, \delta_\alpha\rangle) = \hat{f}_e(\beta)]\!]_{GSB}(\sigma_L^e)$		

   - The proof for the preservation of equality with `null` and equality between two cutpoint-anchored paths is analogous.

3.
Immediate from the definition of f_e; the substitution of actual parameters by formal parameters as defined in Figure 2.5 and Figure 4.7; and the fact that the $h^e = h^c$.

4.

43. $\forall \alpha \in GAccPath_q$:
$[\alpha]_{\sigma_L^c} \in O_c^{passed}$ $\iff$ Definition of O_c^{passed},
$[\alpha]_{A^c} \neq \emptyset \wedge$
$\exists i, 1 \leq i \leq k \colon \exists \delta \in \Delta \colon [\![\alpha = \langle x_i, \delta \rangle]\!]_{LSL}(\sigma_L^c)$ $\iff$ (see proof for Lemma D.2.8(1))
$[\![\alpha \neq \texttt{null}]\!]_{LSL}(\sigma_L^c) \wedge$
$\exists i, 1 \leq i \leq k \colon \exists \delta \in \Delta \colon [\![\alpha = \langle x_i, \delta \rangle]\!]_{LSL}(\sigma_L^c)$ Lemma D.2.1(1)

44. $\forall \alpha \in GAccPath_q$:
$[\![\hat{f}(\alpha)]\!]_G(s_G^c) \in L^{reach}$ $\iff$ By definition of L^{reach}
$[\![\hat{f}(\alpha)]\!]_G(s_G^c) \in L^c \ \wedge$
$\exists i, 1 \leq i \leq kv \colon \exists \delta \in \Delta \colon [\![\hat{f}(\alpha) = \langle x_i, \delta \rangle]\!]_{GSB}(s_G^c)$ $\iff$ Def. of equality with `null`
$[\![\hat{f}(\alpha) \neq \texttt{null}]\!]_{GSB}(s_G^c) \ \wedge$
$\exists i, 1 \leq i \leq k . \exists \delta \in \Delta . [\![\hat{f}(\alpha) = \langle x_i, \delta \rangle]\!]_{GSB}(s_G^c)$

45. $\forall \alpha \in GAccPath_q \colon \forall i, 1 \leq i \leq k \colon \forall \delta \in \Delta$:
$[\![\alpha \neq \texttt{null}]\!]_{LSL}(\sigma_L^c) \ \wedge \ [\![\alpha = \langle x_i, \delta \rangle]\!]_{LSL}(\sigma_L^c)$ $\iff$ $\sigma_L^c \propto_f s_G^c$
$[\![\hat{f}(\alpha) \neq \texttt{null}]\!]_{GSB}(s_G^c) \ \wedge \ [\![\hat{f}(\alpha) = \langle x_i, \delta \rangle]\!]_{GSB}(s_G^c)$

46. $\forall \alpha \in GAccPath_q$:
$[\alpha]_{\sigma_L^c} \in O_c^{passed} \iff [\![\hat{f}(\alpha)]\!]_G(A^c) \in L^{reach}$ $43 - 45$

5.

Immediate from the following facts:

1. formal parameters are not assigned;
2. $L^e \subseteq L^x$; and
3. by definition of $[\![\cdot]\!]_G$, for any $cpl \in CPL^e$,

$$[\![\hat{f}_e(\langle cpl, \epsilon \rangle)]\!]_G(s_G^e) = f_e(cpl) = [\![\hat{f}_e(\langle cpl, \epsilon \rangle)]\!]_G(s_G^x).$$

The following lemma shows that context-aware equivalence at the call-site is preserved at the corresponding return-site for access paths that do not traverse the local-heap of the invoked procedure (1–2). Furthermore, it asserts that if the exit-states are context-aware equivalent w.r.t f_e (as defined in the previous lemma), then the return states are also context-aware equivalent w.r.t. f_e (3). This is the main lemma used in the proof of Theorem D.2.9.

Lemma D.2.8 (Context-aware equivalence at return sites) *Let σ_L^c, s_G^c, f, $y = p(x_1, \ldots, x_k)$, p, σ_L^e, σ_L^x, σ_L^r, O_c^{args}, O_c^{passed}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, $bind_{call}$, s_G^e, s_G^x, L^{reach}, and f_e be as in Lemma D.2.7. The following holds,*

1. *$\forall \alpha \in GAccPath_q$ if $[\alpha]_{A^r} \neq \emptyset \wedge [\alpha]_{A^r} \in A^c \setminus O_c^{passed}$ then (i) $[\alpha]_{A^r} = [\alpha]_{A^c}$ and (ii) $[\![\hat{f}(\alpha)]\!]_G(s_G^r) = [\![\hat{f}(\alpha)]\!]_G(s_G^c) \notin L^{reach}$.*
2. *For any $o \in O_c^{passed}$, for any $\alpha_q \in Bypass(O_c^{passed})$ o, $[\![\hat{f}(\alpha_q)]\!]_G(s_G^c) = [\![\hat{f}(\alpha_q)]\!]_G(s_G^r)$.*
3. *If $\sigma_L^x \propto_{f_e} s_G^x$ then $\sigma_L^r \propto_f s_G^r$.*

Proof:

1.

47.	$[\alpha]_{A^r} \in A^c \setminus (O_c^{passed} \cup \{\emptyset\})$	Assumption
48.	$[\alpha]_{A^c} = [\alpha]_{A^r}$	Admissibility of σ_L^c
49.	$[\alpha]_{A^c} \notin \mathit{RObjs}(O_c^{args})$	$47-48$, $O_c^{passed} = \mathit{RObjs}(O_c^{args})$
50.	$\forall o \in A^c, \forall o' \in O_c^{args}, \forall \delta \in \Delta, o'.\delta \subseteq o \implies \alpha \notin o$	49, def. of $\mathit{RObjs}(O_c^{args})$
51.	$\forall o \in A^c, \forall i, 1 \leq i \leq k, \forall \delta \in \Delta,$ $([x_i]_{A^c} \neq \emptyset \wedge [x_i]_{A^c}.\delta \subseteq o) \implies \alpha \notin o$	50, def. of O_c^{args}
52.	$[\alpha]_{A^c} \neq \emptyset$	$47-48$
53.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\alpha]_{A^c} \neq [\langle x_i, \delta \rangle]_{A^c}$	51, 52
54.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\![\alpha \neq \langle x_i, \delta \rangle]\!]_{LSL}(\sigma_L^c)$	53, def. of equality
55.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\![\hat{f}(\alpha) \neq \langle x_i, \delta \rangle]\!]_{GSB}(s_G^c)$	54, $\sigma_L^c \propto_f s_G^c$, $\hat{f}(\langle x_i, \delta \rangle) = \langle x_i, \delta \rangle$
56.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\![\hat{f}(\alpha)]\!]_G(s_G^c) \neq [\![\langle x_i, \delta \rangle]\!]_G(s_G^c)$	55, def. of $[\![\cdot = \cdot]\!]_{GSB}$
57.	$[\![\hat{f}(\alpha)]\!]_G(s_G^c) \notin L^{reach}$	56, def. of L^{reach}
58.	$[\![\hat{f}(\alpha)]\!]_G(s_G^c) = [\![\hat{f}(\alpha)]\!]_G(s_G^r)$	57, Lemma B.2.5

2.
Immediate from the definition of *Bypass*, Lemma D.2.8(1), Lemma B.2.5(2), and the fact that a callee cannot modify the value of pending variables.

3

Let $\alpha = \langle r_\alpha, \delta_\alpha \rangle$, $\beta = \langle r_\beta, \delta_\beta \rangle$, and $\gamma = \langle r_\gamma, \delta_\gamma \rangle$ be any generalized access paths of procedure p. We show that

1. $[\![\alpha = \beta]\!]_{LSL}(\sigma_L^r) \implies [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G^r)$ and
2. $[\![\gamma = \texttt{null}]\!]_{LSL}(\sigma_L^r) \implies [\![\hat{f}(\gamma) = \texttt{null}]\!]_{GSB}(s_G^r)$.

The proof of the other direction, (i.e., that $[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G^r) \implies [\![\alpha = \beta]\!]_{LSL}(\sigma_L^r)$ and $[\![\hat{f}(\gamma) = \texttt{null}]\!]_{GSB}(s_G^r) \implies [\![\gamma = \texttt{null}]\!]_{LSL}(\sigma_L^r)$) is analogous, and it is not shown.

The proof is done by case analysis.

1. Proving $[\![\alpha = \beta]\!]_{LSL}(\sigma_L^r) \Rightarrow [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G^r)$
   a. Assuming $[\![\alpha \neq \texttt{null}]\!]_{LSL}(\sigma_L^r)$ and $[\![\beta \neq \texttt{null}]\!]_{LSL}(\sigma_L)$ and
      1. $[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \in A^c \setminus O_c^{passed}$.
      2. $[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \notin A^c \setminus O_c^{passed}$.
      3. $[\alpha]_{A^r} \notin A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \in A^c \setminus O_c^{passed}$.
      4. $[\alpha]_{A^r} \notin A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \notin A^c \setminus O_c^{passed}$.
   b. Assuming $[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L^r)$ and $[\![\beta = \texttt{null}]\!]_{LSL}(\sigma_L^r)$.
2. Proving $[\![\gamma = \texttt{null}]\!]_{LSL}(\sigma_L^r) \Rightarrow [\![\hat{f}(\gamma) = \texttt{null}]\!]_{GSB}(s_G^r)$
   a. $r_\gamma = y$.
   b. $r_\gamma \neq y$ and
      1. $\forall \gamma' \leq \gamma, [\gamma']_{A^r} \in (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$.
      2. $\exists \gamma' \leq \gamma, [\gamma']_{A^r} \notin (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$.

Case 1(a)1:

59.	$[\alpha]_{A^r} \neq \emptyset$	$[\![\alpha \neq \texttt{null}]\!]_{LSL}(\sigma_L^r), Lemma\ D.2.1(1)$
60.	$[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$	Assumption
61.	$[\alpha]_{A^c} = [\alpha]_{A^r}$	$59-61$, Lemma D.2.8(1)
62.	$[\![\hat{f}(\alpha)]\!]_G(s_G^c) = [\![\hat{f}(\alpha)]\!]_G(s_G^r)$	$59-61$, Lemma D.2.8(1)
63.	$[\beta]_{A^c} = [\beta]_{A^r} \neq \emptyset$	Analogous to $59-61$
64.	$[\![\hat{f}(\beta)]\!]_G(s_G^c) = [\![\hat{f}(\beta)]\!]_G(s_G^r)$	Analogous to 62
65.	$[\alpha]_{A^r} = [\beta]_{A^r}$	$[\![\alpha = \beta]\!]_{LSL}(\sigma_L^r), Lemma\ D.2.1(2)$
66.	$[\alpha]_{A^c} = [\beta]_{A^c}$	61, 63, 65
67.	$[\![\alpha = \beta]\!]_{LSL}(\sigma_L^c)$	66, $Lemma\ D.2.1(2)$
68.	$[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G^c)$	67, $\sigma_L^c \propto_f s_G^c$
69.	$[\![\hat{f}(\alpha)]\!]_G(s_G^c) = [\![\hat{f}(\beta)]\!]_G(s_G^c)$	By def. of equality
70.	$[\![\hat{f}(\alpha)]\!]_G(s_G^r) = [\![\hat{f}(\beta)]\!]_G(s_G^r)$	62, 64, 69
71.	$[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G^r)$	70, def. of equality

Case 1(a)2: This case is impossible.

72.	$[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$, $[\alpha]_{A^r} \neq \emptyset$	See $Case\ 1(a)1$.
73.	$[\beta]_{A^r} \neq \emptyset$	$[\![\beta \neq \texttt{null}]\!]_{LSL}(\sigma_L^r)$, $Lemma\ D.2.1(1)$
74.	$[\beta]_{A^r} \in map(sub(bind_{ret}))\ A^x \setminus \{\emptyset\}$	73, $[\beta]_{A^r} \notin A^c \setminus O_c^{passed}$ $Lemma\ D.2.3(2)$
75.	$(A^c \setminus O_c^{passed}) \cap map(sub(bind_{ret}))\ A^x \subseteq \{\emptyset\}$	$Lemma\ D.2.2(2)$
76.	$[\alpha]_{\sigma_L^r} = [\beta]_{\sigma_L^r}$	$[\![\alpha = \beta]\!]_{LSL}(\sigma_L^r)$, $Lemma\ D.2.1(2)$
77.	Contradiction	$72, 73, 74-76$

Case 1(a)3: This case is also impossible (see proof for Case 1(a)2).

Case 1(a)4:

78. $[\alpha]_{A^r} \neq \emptyset$ — $[\![\alpha \neq \texttt{null}]\!]_{LSL}(\sigma_L^r)$, *Lemma D*.2.1(1)

79. $[\alpha]_{A^r} \in (map(sub(bind_{ret}))\ A^x) \setminus \{\emptyset\}$ — 78, Assumption, *Lemma D*.2.3(2)

80. $[\beta]_{A^r} \neq \emptyset$ — $[\![\beta \neq \texttt{null}]\!]_{LSL}(\sigma_L^r)$, *Lemma D*.2.1(1)

81. $[\beta]_{A^r} \in (map(sub(bind_{ret}))\ A^x) \setminus \{\emptyset\}$ — 80, Assumption, *Lemma D*.2.3(2)

82. $[\alpha]_{A^r} = [\beta]_{A^r}$ — $[\![\alpha = \beta]\!]_{LSL}(\sigma_L^r)$, *Lemma D*.2.1(2)

83. $\exists! o \in A^x, [\alpha]_{A^r} = [\beta]_{A^r} = sub(bind_{ret})\ o \neq \emptyset$ — $78 - 82$, Def. of *map*, *Lemma D*.2.3(4)

84. Let o be the unique object in A^x which satisfies 83.
Let $\alpha_0, r_1^\alpha, \delta_1^\alpha$ be the unique values determined by *Lemma D*.2.3(6) for o and α.
Let $\beta_0, r_1^\beta, \delta_1^\beta$ be the unique values determined by *Lemma D*.2.3(6) for o and β.

85. $\alpha_0 \neq \langle y, \epsilon\rangle \implies$
$\alpha_0 \in Bypass(O_c^{passed})\ [\alpha_0]_{\sigma_L^c}$, $[\alpha_0]_{A^c} \in O_c^{args} \cup O_c^{cp}$,
$[\alpha_0]_{A^c} \in O_c^{args} \implies r_1^\alpha = bind_{args}\ [\alpha_0]_{A^c} \neq \emptyset$
$[\alpha_0]_{A^c} \in O_c^{cp} \implies r_1^\alpha = bind_{cp}\ [\alpha_0]_{A^c} \neq \emptyset$ — *Lemma D*.2.3(6)

86. $[\![\hat{f}(\alpha_0)]\!]_G(s_G^r) = \begin{cases} [\![\hat{f}(\langle y, \epsilon\rangle)]\!]_G(s_G^r) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}(\alpha_0)]\!]_G(s_G^r) & \alpha_0 \neq \langle y, \epsilon\rangle \end{cases}$ — *Lemma D*.2.3(6)

87. $[\![\hat{f}(\alpha_0)]\!]_G(s_G^r) = \begin{cases} [\![\langle y, \epsilon\rangle]\!]_G(s_G^r) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}(\alpha_0)]\!]_G(s_G^c) & \alpha_0 \neq \langle y, \epsilon\rangle \end{cases}$ — Def. of $\hat{f}$; 85, $O_c^{args} \cup O_c^{cp} \subseteq O_c^{passed}$, *Lemma D*.2.8(2)

88. $[\![\hat{f}(\alpha_0)]\!]_G(s_G^r) = \begin{cases} [\![\langle ret, \epsilon\rangle]\!]_G(s_G^x) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}_e(\alpha_p)]\!]_G(s_G^e) & \alpha_0 \neq \langle y, \epsilon\rangle,\ \alpha_p \in r_1^\alpha \end{cases}$ — Def. of $\mathcal{GSB}$ see Figure 2.5; 85, *Lemma D*.2.7(3)

89. $[\![\hat{f}(\alpha_0)]\!]_G(s_G^r) = \begin{cases} [\![\hat{f}_e(\langle ret, \epsilon\rangle)]\!]_G(s_G^x) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}_e(\alpha_p)]\!]_G(s_G^x) & \alpha_0 \neq \langle y, \epsilon\rangle,\ \alpha_p \in r_1^\alpha \end{cases}$ — Def. of $\hat{f}_e$; 88, *Lemma D*.2.7(5)

90. $\forall \alpha_p \in r_1^\alpha, [\![\hat{f}(\alpha_0)]\!]_G(s_G^r) = [\![\hat{f}_e(\alpha_p)]\!]_G(s_G^x)$ — 89, $\alpha_0 = \langle y, \epsilon\rangle \Rightarrow r_1^\alpha = \{\langle ret, \epsilon\rangle\}$

91. $[\![\hat{f}(\alpha)]\!]_G(s_G^r) = [\![\langle [\![\hat{f}(\alpha_0)]\!]_G(s_G^r), \delta_1^\alpha\rangle]\!]_G(s_G^r)$ — $\alpha = \alpha_0.\delta_1^\alpha$, *Lemma B*.2.6(1), *Lemma D*.2.6

92. $\forall \beta_p \in r_1^\beta, [\![\hat{f}(\beta_0)]\!]_G(s_G^r) = [\![\hat{f}_e(\beta_p)]\!]_G(s_G^x)$ — Analogous to $85 - 90$

93. $[\![\hat{f}(\beta)]\!]_G(s_G^r) = [\![\langle [\![\hat{f}(\beta_0)]\!]_G(s_G^r), \delta_1^\beta\rangle]\!]_G(s_G^r)$ — Analogous to 91

94. $r_1^\alpha.\delta_1^\alpha \subseteq o$, $r_1^\beta.\delta_1^\beta \subseteq o$ — 84, *Lemma D*.2.3(6)

95. $\forall \alpha_p \in r_1^\alpha, \forall \beta_p \in r_1^\beta, [\![\alpha_p.\delta_1^\alpha = \beta_p.\delta_2^\beta]\!]_{LSL}(s_G^x)$ — 94, *Lemma D*.2.1(2), Admissibility of σ_L^x

96. $\sigma_L^x \propto_{f_e} s_G^x$ — Assumption

97. $\forall \alpha_p \in r_1^\alpha, \forall \beta_p \in r_1^\beta, [\![\hat{f}_e(\alpha_p.\delta_1^\alpha) = \hat{f}_e(\beta_p.\delta_2^\beta)]\!]_{GSB}(s_G^x)$ — $95 - 96$

98. $\forall \alpha_p \in r_1^\alpha, \forall \beta_p \in r_1^\beta, [\![\hat{f}_e(\alpha_p.\delta_1^\alpha)]\!]_G(s_G^x) = [\![\hat{f}_e(\beta_p.\delta_2^\beta)]\!]_G(s_G^x)$ — 97, Def. of equality

99. $\forall \alpha_p \in r_1^\alpha, \forall \beta_p \in r_1^\beta,$
$[\![\langle [\![\hat{f}_e(\alpha_p)]\!]_G(s_G^x), \delta_1^\alpha\rangle]\!]_G(s_G^x) = [\![\langle [\![\hat{f}_e(\beta_p)]\!]_G(s_G^x), \delta_1^\beta\rangle]\!]_G(s_G^x)$ — 98

100. $[\![\langle [\![\hat{f}(\alpha_0)]\!]_G(s_G^r), \delta_1^\alpha\rangle]\!]_G(s_G^x) = [\![\langle [\![\hat{f}(\beta_0)]\!]_G(s_G^r), \delta_1^\beta\rangle]\!]_G(s_G^x)$ — 90, 92, 99, 85 ($r_1^\alpha \neq \emptyset$, $r_1^\beta \neq \emptyset$)

101. $[\![\langle [\![\hat{f}(\alpha_0)]\!]_G(s_G^r), \delta_1^\alpha\rangle]\!]_G(s_G^r) = [\![\langle [\![\hat{f}(\beta_0)]\!]_G(s_G^r), \delta_1^\beta\rangle]\!]_G(s_G^r)$ — 100, $h^r = h^x$, Def. of $[\![\cdot]\!]_G$

102. $[\![\hat{f}(\alpha)]\!]_G(s_G^r) = [\![\hat{f}(\beta)]\!]_G(s_G^r)$ — 91, 93, 101

103. $[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G^r)$ — 102, Def. of equality

Case 1b:

104.	$[\![\alpha = \texttt{null}]\!]_{LSL}(\sigma_L^r)$	Assumption
105.	$[\![\beta = \texttt{null}]\!]_{LSL}(\sigma_L^r)$	Assumption
106.	$[\![\hat{f}(\alpha) = \texttt{null}]\!]_{GSB}(s_G^r)$	104, $\sigma_L^c \propto_f s_G^c$, $Case\ 2$
107.	$[\![\hat{f}(\beta) = \texttt{null}]\!]_{GSB}(s_G^r)$	105, $\sigma_L^c \propto_f s_G^c$, $Case\ 2$
108.	$[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s_G^r)$	106, 107

Case 2a:

109.	$[\![\langle y, \delta_\gamma\rangle = \texttt{null}]\!]_{LSL}(\sigma_L^r)$	By assumption
110.	$[\![\langle ret, \delta_\gamma\rangle = \texttt{null}]\!]_{LSL}(\sigma_L^x)$	$Lemma\ D.2.3(5a)$
111.	$\sigma_L^x \propto_{f_e} s_G^x$	$\sigma_L^c \propto_{f_e} s_G^c$, the induction assumption
112.	$[\![\hat{f}_e(\langle ret, \delta_\gamma\rangle) = \texttt{null}]\!]_{GSB}(s_G^x)$	110, 111
113.	$[\![\langle ret, \delta_\gamma\rangle = \texttt{null}]\!]_{GSB}(s_G^x)$	112, Def. of $\hat{f}_e$
114.	$[\![\langle y, \delta_\gamma\rangle = \texttt{null}]\!]_{GSB}(s_G^r)$	$h^r = h^c, \rho^r(y) = \rho^x(ret)$

Case 2(b)1:

115.	$\forall\gamma' \leq \gamma, [\gamma']_{A^r} \in (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$	Assumption
116.	$A^r = (A^c \setminus O_c^{passed}) \cup map(sub(bind_{ret}))\ A^x$	Def. of $\mathcal{LSL}$ (see Figure 4.7).
117.	$\forall\gamma' \leq \gamma, [\gamma']_{\sigma_L^c} \notin O_c^{passed}$	116, $Lemma\ D.2.3(3), Lemma\ D.2.2(1)$
118.	$[\![\gamma = \texttt{null}]\!]_{LSL}(\sigma_L^c)$	115, 117, $\gamma \leq \gamma, Lemma\ D.2.3(5b)$
119.	$[\![\hat{f}(\gamma) = \texttt{null}]\!]_{GSB}(\sigma_L^c)$	118, $\sigma_L^c \propto_f s_G^c$
	We continue with case analysis w.r.t. the value of $[\![\langle r_\gamma, \epsilon\rangle]\!]_G(s_G^c)$	
•	Assume $[\![\langle r_\gamma, \epsilon\rangle]\!]_G(s_G^c) = \textit{null}$	
120.	$[\![\gamma = \texttt{null}]\!]_{LSL}(\sigma_L^r)$	$\forall\gamma' \leq \gamma [\![\gamma' \neq \texttt{null}]\!]_{LSL}(\sigma_L^r)$
121.	$r_\gamma \in V_q \setminus \{y\}$	120, $Lemma\ D.2.1(6)$, Assumption $(r_\gamma \neq y)$
122.	$[\![\langle r_\gamma, \epsilon\rangle = \texttt{null}]\!]_{LSL}(\sigma_L^c)$	$120 - 121$, $Lemma\ D.2.3(5a)$
123.	$[\![\hat{f}(\langle r_\gamma, \epsilon\rangle) = \texttt{null}]\!]_{GSB}(s_G^c)$	122, $\sigma_L^c \propto_f s_G^c$
124.	$[\![\langle r_\gamma, \epsilon\rangle = \texttt{null}]\!]_{GSB}(s_G^c)$	Def. of $\hat{f}$
125.	$[\![\langle r_\gamma, \epsilon\rangle = \texttt{null}]\!]_{GSB}(s_G^r)$	Def. of $\mathcal{GSB}$
126.	$[\![\gamma = \texttt{null}]\!]_{GSB}(s_G^r)$	125, $\langle r_\gamma, \epsilon\rangle \leq \gamma$, $Lemma\ B.2.6(2a)$
•	Assume $[\![\langle r_\gamma, \epsilon\rangle]\!]_G(s_G^c) \neq \textit{null}$	
127.	$\exists! s \in \mathcal{F}, \exists!\gamma'.s \leq \gamma, [\![\hat{f}(\gamma') \neq \texttt{null}]\!]_{GSB}(\sigma_L^c) \wedge \forall\gamma'', \gamma'.s \leq \gamma'', [\![\hat{f}(\gamma'') = \texttt{null}]\!]_{GSB}(\sigma_L^c)$	119, $Lemma\ B.2.6(2a - 2b)$, $Lemma\ D.2.6$
128.	Let f and γ' be the unique values satsfying 127	
129.	$[\![\hat{f}(\gamma')]\!]_G(\sigma_L^c) \notin L^{reach}$	117, 128, $Lemma\ D.2.7(4)$
130.	$[\![\hat{f}(\gamma')]\!]_G(\sigma_L^c) \in L^c \setminus L^{reach}$	129, $[\![\hat{f}(\gamma') \neq \texttt{null}]\!]_{GSB}(\sigma_L^c)$
131.	$[\![\hat{f}(\gamma').s]\!]_G(\sigma_L^c) = [\![\hat{f}(\gamma').s]\!]_G(\sigma_L^r)$	130, $Lemma\ B.2.5$, $Lemma\ D.2.6$
132.	$[\![\hat{f}(\gamma').s]\!]_G(\sigma_L^r) = \textit{null}$	$127 - 128$, 131, $Lemma\ D.2.6$
133.	$[\![\hat{f}(\gamma').s = \texttt{null}]\!]_{GSB}(\sigma_L^r)$	132, Def. of equality w. `null`
134.	$[\![\hat{f}(\gamma) = \texttt{null}]\!]_{GSB}(\sigma_L^r)$	128, 133, $Lemma\ B.2.6(2a)$, $Lemma\ D.2.6$

Case 2(b)2:

135.	$[\gamma]_{\sigma_L^r} = \emptyset$	$Lemma\ D.2.1(1)$ $(\llbracket \gamma = \texttt{null} \rrbracket_{LSL}(\sigma_L^r))$
136.	$\exists \gamma' \leq \gamma, [\gamma']_{A^r} \notin (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$	Assumption
137.	$\exists \gamma', \gamma' < \gamma, [\gamma']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \setminus \{\emptyset\}$, $\forall \gamma'', \gamma' \leq \gamma'' \implies [\gamma'']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \cup \{\emptyset\}$	$135 - 136$, $Lemma\ D.2.3(2)$
138.	$\exists \gamma', \gamma' < \gamma, [\gamma']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \setminus \{\emptyset\}$, $\forall \gamma'', \gamma' \leq \gamma'' \implies [\gamma'']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \cup \{\emptyset\}$	137, $\llbracket \gamma = \texttt{null} \rrbracket_{LSL}(\sigma_L^r)$, $Lemma\ D.2.1(1)$
139.	$\exists s \in FID, \exists \gamma', \gamma'.s \leq \gamma \wedge [\gamma']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \wedge$ $\llbracket \gamma' \neq \texttt{null} \rrbracket_{LSL}(\sigma_L^r) \wedge \forall \gamma'', \gamma'.s \leq \gamma'' \implies \llbracket \gamma'' = \texttt{null} \rrbracket_{LSL}(\sigma_L^r)$	138, $Lemma\ D.2.1(1,3,4)$
140.	$\exists s \in FID, \exists \gamma', \gamma'.s \leq \gamma, \exists o \in A^x,\ [\gamma']_{\sigma_L^r} = sub(bind_{ret})\ o \wedge$ $\llbracket \gamma' \neq \texttt{null} \rrbracket_{LSL}(\sigma_L^r) \wedge \llbracket \gamma'.s = \texttt{null} \rrbracket_{LSL}(\sigma_L^r)$	139, Def. of map
141.	Let γ', s, o be the unique values satisfying 140	
142.	let $\gamma_0', r_1^{\gamma'}, \delta_1^{\gamma'}$ be the unique values determined by $Lemma\ D.2.3(6)$ for o and γ'.	
143.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \gamma_p.\delta_1^{\gamma'} s = \texttt{null} \rrbracket_{LSL}(\sigma_L^x)$	$Lemma\ D.2.3(5c)$
144.	$\sigma_L^x \propto_{f_e} s_G^x$	Assumption
145.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \hat{f}_e(\gamma_p.\delta_1^{\gamma'} s) = \texttt{null} \rrbracket_{GSB}(\sigma_L^x)$	$143 - 144$
146.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \hat{f}_e(\gamma_p.\delta_1^{\gamma'} s) \rrbracket_G(\sigma_L^x) = \textit{null}$	145
147.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \llbracket \hat{f}_e(\gamma_p.\delta_1^{\gamma'}) \rrbracket_G(\sigma_L^x).s \rrbracket_G(\sigma_L^x) = \textit{null}$	146, $Lemma\ D.2.6$
148.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \llbracket \llbracket \hat{f}_e(\gamma_p) \rrbracket_G(\sigma_L^x).\delta_1^{\gamma'} \rrbracket_G(\sigma_L^x).s \rrbracket_G(\sigma_L^x) = \textit{null}$	147, $Lemma\ D.2.6$
149.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \hat{f}(\gamma_0) \rrbracket_G(s_G^r) = \llbracket \hat{f}_e(\gamma_p) \rrbracket_G(s_G^x),\ r_1^{\gamma'} \neq \emptyset$	See proof for $Case\ 1(a)4$
150.	$\llbracket \llbracket \llbracket \hat{f}(\gamma_0) \rrbracket_G(\sigma_L^r).\delta_1^{\gamma'} \rrbracket_G(\sigma_L^x).s \rrbracket_G(\sigma_L^x) = \textit{null}$	$148 - 149$
151.	$\llbracket \llbracket \llbracket \hat{f}(\gamma_0) \rrbracket_G(\sigma_L^r).\delta_1^{\gamma'} \rrbracket_G(\sigma_L^r).s \rrbracket_G(\sigma_L^r) = \textit{null}$	150, $h^x = h^r$ (See Figure 4.7)
152.	$\llbracket \gamma'.s \rrbracket_G(\sigma_L^r) = \textit{null}$	151, $Lemma\ D.2.6$
153.	$\llbracket \gamma'.s = \texttt{null} \rrbracket_{GSB}(\sigma_L^r)$	152, Def. of $\llbracket \cdot \rrbracket_G$
154.	$\llbracket \gamma = \texttt{null} \rrbracket_{GSB}(\sigma_L^r)$	$\gamma'.s \leq \gamma$, $Lemma\ B.2.6(2a)$

Theorem D.2.9 (Context-aware Equivalence Preservation) *Let p be a procedure. Let $\sigma_L \in \Sigma_L^p$ and $s_G \in \mathcal{S}_G^p$ be context-aware equivalent states w.r.t. to a renaming function f, i.e., $\sigma_L \propto_f s_G$. Let st be an arbitrary statement in p. The following holds:*

1. *For any state $\sigma_L' \in \Sigma_L^p$ such that $\langle st, \sigma_L \rangle \overset{LSL}{\rightsquigarrow} \sigma_L'$, there exists a state $s_G' \in \mathcal{S}_G^p$ such that:*
   - $\langle st, s_G \rangle \overset{GSB}{\rightsquigarrow} s_G'$, *and*
   - $\sigma_L' \propto_f s_G'$.
2. *For any state $s_G' \in \mathcal{S}_G^p$ such that $\langle st, s_G \rangle \overset{GSB}{\rightsquigarrow} s_G'$, there exists a state $\sigma_L' \in \Sigma_L^p$ such that:*
   - $\langle st, \sigma_L \rangle \overset{LSL}{\rightsquigarrow} \sigma_L'$, *and*
   - $\sigma_L' \propto_f s_G'$.

Proof: Let q be a procedure. Let $\sigma_L = \langle \mathit{CPL}, A \rangle \in \Sigma_L^q$ and $s_G = \langle L, \rho, h \rangle \in \mathcal{S}_G^q$ context-aware equivalent states w.r.t. to a renaming function f, i.e., $\sigma_L \propto_f s_G$.

We prove (i) and (ii) simultaneously using an induction on the shape of the derivation tree. The proof is done by case analysis of the statement in the transition which labels the root of the derivation tree.

Base case: The transition which labels the root of the derivation tree contains an atomic statement, i.e., the derivation tree is a leaf. Thus, we only need to show that the states that result by executing the same (atomic) statement in σ_L and s_G are also context-aware equivalent w.r.t. to f. The proof is done by a case analysis.

x=null The axiom for this statement has no side-condition, thus this statement is guaranteed to terminate in any state. In particular, it is true that

$\exists \sigma'_L \in \Sigma^q_L,$ s.t. $\langle \mathtt{x = null}, \sigma_L \rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$ and $\exists s'_G \in \mathcal{S}^q_G,$ s.t. $\langle \mathtt{x = null}, s_G \rangle \overset{GSB}{\rightsquigarrow} s'_G$.
By definition, $\sigma'_L = \langle CPL, rem(A, \{x\}) \rangle = \langle CPL, \{(map(\lambda o.o \setminus \{x\}.\Delta)\ A) \setminus \{\emptyset\} \rangle = \langle CPL, \{a \setminus x.\Delta \mid a \in A\} \setminus \{\emptyset\} \rangle$ and $s'_G = \langle L, \rho[x \mapsto \mathit{null}], h \rangle$.

Let $\alpha = \langle r_\alpha, \delta_\alpha \rangle$, $\beta = \langle r_\beta, \delta_\beta \rangle$, and $\gamma = \langle r_\gamma, \delta_\gamma \rangle$ be generalized access paths of procedure q.

$$
\begin{array}{ll}
[\![\alpha = \beta]\!]_L(\sigma'_L) & \iff \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\} \setminus \{\emptyset\}, \alpha \in a' \iff \beta \in a' & \iff \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\}, \alpha \in a' \iff \beta \in a' & \iff \\
\left\{ \begin{array}{ll} r_\alpha \neq x, r_\beta \neq x, \forall a \in A, \alpha \in a \iff \beta \in a & \text{or} \\ r_\alpha = x, r_\beta \neq x, \forall a \in A, \beta \notin a & \text{or} \\ r_\alpha \neq x, r_\beta = x, \forall a \in A, \alpha \notin a & \text{or} \\ r_\alpha = x, r_\beta = x \end{array} \right. & \iff \\
\left\{ \begin{array}{ll} r_\alpha \neq x, r_\beta \neq x, [\![\alpha = \beta]\!]_L(\sigma_L) & \text{or} \\ r_\alpha = x, r_\beta \neq x, [\![\beta = \mathtt{null}]\!]_L(\sigma_L) & \text{or} \\ r_\alpha \neq x, r_\beta = x, [\![\alpha = \mathtt{null}]\!]_L(\sigma_L) & \text{or} \\ r_\alpha = x, r_\beta = x \end{array} \right. & \iff \quad (\sigma_L \propto_f s_G) \\
\left\{ \begin{array}{ll} r_\alpha \neq x, r_\beta \neq x, [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(s_G) & \text{or} \\ r_\alpha = x, r_\beta \neq x, [\![\hat{f}(\beta) = \mathtt{null}]\!]_G(s_G) & \text{or} \\ r_\alpha \neq x, r_\beta = x, [\![\hat{f}(\alpha) = \mathtt{null}]\!]_G(s_G) & \text{or} \\ r_\alpha = x, r_\beta = x \end{array} \right. & \iff \\
[\![\hat{f}(\alpha)]\!]_G(s'_G) = [\![\hat{f}(\beta)]\!]_G(s'_G) & \iff \\
[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_{GSB}(s'_G) &
\end{array}
$$

$$
\begin{array}{ll}
[\![\gamma = \mathtt{null}]\!]_L(\sigma'_L) & \iff \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\} \setminus \{\emptyset\}, \gamma \notin a' & \iff \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\}, \gamma \notin a' & \iff \\
\left\{ \begin{array}{ll} r_\gamma \neq x, \forall a \in A, \gamma \notin a & \text{or} \\ r_\gamma = x \end{array} \right. & \iff \\
\left\{ \begin{array}{ll} r_\gamma \neq x, [\![\gamma = \mathtt{null}]\!]_L(\sigma_L) & \text{or} \\ r_\gamma = x \end{array} \right. & \iff \quad (\sigma_L \propto_f s_G) \\
\left\{ \begin{array}{ll} r_\gamma \neq x, [\![\hat{f}(\gamma) = \mathtt{null}]\!]_G(s_G) & \text{or} \\ r_\gamma = x \end{array} \right. & \iff \\
[\![\hat{f}(\gamma)]\!]_G(s'_G) = \mathit{null} & \iff \\
[\![\hat{f}(\gamma) = \mathtt{null}]\!]_{GSB}(s'_G) &
\end{array}
$$

x=y Analogous to [x=null].

x=y.f Analogous to [x=null].

x.f=null Analogous to [x=null].

x.f=y Analogous to [x=null].

x=alloc t Analogous to [x=null].

Induction step (intraprocedural): The transition labeling the root of the derivation tree contains a non-atomic intraprocedural control statement. Thus, the induced derivation tree is not a leaf. The proof is done by a case analysis.

seq Assume that $st = st_1; st_2$ and that $\langle st_1; st_2, \sigma_L\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$.

155.	$\langle st_1; st_2, \sigma_L\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$	Assumption
156.	$\exists \sigma''_L.\langle st_1, \sigma_L\rangle \overset{LSL}{\rightsquigarrow} \sigma''_L \wedge \langle st_2, \sigma''_L\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L$	Def. of [seq] in $\mathcal{LSL}$
157.	$\exists \sigma''_L.\exists s''_G.\langle st_1, \sigma_L\rangle \overset{LSL}{\rightsquigarrow} \sigma''_L \wedge \langle st_2, \sigma''_L\rangle \overset{LSL}{\rightsquigarrow} \sigma'_L \wedge \langle st_1, s_G\rangle \overset{GSB}{\rightsquigarrow} s''_G \wedge \sigma''_L \propto_f s''_G$	$\sigma_L \propto_f s_G$ Induction assumption for st_1, σ_L, and s_G
158.	$\exists s''_G.\exists s'_G.\langle st_1, s_G\rangle \overset{GSB}{\rightsquigarrow} s''_G \wedge \langle st_2, s''_G\rangle \overset{GSB}{\rightsquigarrow} s'_G \wedge \sigma'_L \propto_f s'_G$	$\sigma''_L \propto_f s''_G$ Induction assumption for st_2, σ''_L, and s''_G
159.	$\exists s'_G.\langle st_1; st_2, s_G\rangle \overset{GSB}{\rightsquigarrow} s'_G \wedge \sigma'_L \propto_f s'_G$	Def. of [seq] in $\mathcal{GSB}$

The proof in the other direction is analogous.

if-tt Analogous to [seq].

if-ff Analogous to [seq].

while Analogous to [seq].

Induction step (interprocedural):

The transition labeling the root of the derivation tree contains a procedure call. Thus, the induced derivation tree is not a leaf. Without loss of generality, assume that the invocation is `y=p(`$x_1, \ldots, x_k$`)`. To simplify notations, we assume that $\sigma^c_L = \sigma_L$ and that $s^c_G = s_G$.

Assume that $\langle y = p(x_1, \ldots, x_k), \sigma^c_L\rangle \overset{LSL}{\rightsquigarrow} \sigma^r_L$.

160.	$\langle y = p(x_1, \ldots, x_k), \sigma^c_L\rangle \overset{LSL}{\rightsquigarrow} \sigma^r_L$	Assumption
161.	$\exists \sigma^e_L, \sigma^x_L \in \Sigma^p_L, \sigma^r_L \in \Sigma^q_L, s.t.\ \langle \textit{body of p}, \sigma^e_L\rangle \overset{LSL}{\rightsquigarrow} \sigma^x_L$; and σ^e_L, σ^x_L and σ^r_L are as defined in Figure 4.7	Def. of procedure call in $\mathcal{LSL}$
162.	Let s^e_G be that state that arise at the entry to p when p is invoked at s^c_G	Such s^e_G exists because in $\mathcal{GSB}$ there are no side-conditions for procedure calls
163.	Let f_e be that renaming function defined as in Lemma D.2.8 for σ^e_L and s^e_G	
164.	$\sigma^e_L \propto_{f_e} s^e_G$	162, 163, Lemma D.2.7(2)
165.	$\exists s^x_G \in \mathcal{S}^q_G.\langle \textit{body of p}, s^e_G\rangle \overset{GSB}{\rightsquigarrow} s^x_G \wedge \sigma^x_L \propto_{f_e} s^x_G$	161, 164, Induction assumption for σ^e_L, s^e_G, and f_e
166.	$\exists s^r_G \in \mathcal{S}^q_G.\langle y = p(x_1, \ldots, x_k), s^c_G\rangle \overset{GSB}{\rightsquigarrow} s^r_G\ s.t.$ s^r_G is as defined in Figure 2.5 for s^c_G and s^x_G	165
167.	$\sigma^r_L \propto_f s^r_G$	164, 165, Lemma D.2.8(3)

The proof in the other direction is analogous.

D.3 Formal Properties of the $\mathcal{LCP}$ Semantics

- Section D.3.1 provides additional formal details pertaining to the concrete semantics presented in Section 4.8.2.

D.3.1 Formal Specification of the Operational Semantics

This appendix provides additional formal details. In Section D.3.1.1 we define the notion of garbage collected *2-valued* logical structures. In Section D.3.1.2 we extend the logic of [SRW02] to support extended transitive closure. In Section D.3.1.3 we formally define the operational semantics for procedure calls and returns.

D.3.1.1 Garbage Collected States

Definition D.3.1 (Garbage-Collected 2-Valued Logical Structures) *An* admissible 2-valued *logical structure* $S = \langle U, \iota \rangle$ *representing a local-heap for a procedure* p *in program* P *at a given point in an execution is* ***garbage-collected*** *iff every* object *is reachable from either a variable or from a frozen variable, i.e.,*

$$S \models R_{V_p}(v) \vee \exists v_1, v_2 : isLb_{CP}(v_1) \wedge lbl(v_1, v_2) \wedge (TC\ w_1, w_2 : F(w_1, w_2))(v_2, v)$$

D.3.1.2 Extended Transitive Closure

The logic used in [SRW02] is first order logic with transitive closure. In [SRW02], formulae with transitive closure have only one pair of variables. For example, the bounded variables in the formula $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$ are v_1 and v_2. To define the semantics of the return operation we need to allow a transitive closure formulae with two pairs of free variables, as conducted, e.g., in [Imm99]. (For a formal definition, see Section A.2).

Lemma D.3.2 (Extended Embedding) *Let* φ *be an extended transitive closure formulae. For any* $S \in 2Struct$ *and* $S^\sharp \in 3Struct$ *such that* $S \sqsubseteq^f S^\sharp$ *and for any assignment* Z *it holds that,*

$$[\![\varphi]\!]_2^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S^\sharp}(f \circ Z)$$

D.3.1.3 Operational Semantics

The operational semantics is specified by *predicate-update formulae*: For every predicate p and for every statement st, the value of p in the 2-valued structure S', which results by applying st to S, is defined in terms of a formula evaluated over S.

The predicate-update formulae of the core-predicates for assignments is given in Figure D.1. The value of every core-predicate p after the statement executes, denoted by p', is defined in terms of the core predicate values before the statement executes (denoted without primes). Core predicates whose update formula is not specified, are assumed to be unchanged, i.e., $p'(v_1, \ldots) = p(v_1, \ldots)$.

The operational semantics for object allocation is given in Figure D.2. The operational semantics for procedure invocations is given in Figure D.3 and in Figure D.4.

Definition D.3.3 (Transition Relation) *Let* p *be a procedure. Let* st *is a statement in* p*. The transition relation* $\overset{2}{\rightsquigarrow} \subseteq (2Struct_p \times st) \times 2Struct_p$ *contains* $\langle S, S' \rangle$ *iff (i)* S *and* S' *are garbage-collected* 2-valued *logical structures for procedure* p *and (ii) applying the predicate-update formulae (action) associated with* st *to* S *results in* S'*.*

Statement	**Predicate-update formulae**
`y` $=$ `null`	$y'(v) = 0$
`y` $=$ `x`	$y'(v) = x(v)$
`y` $=$ `x.f`	$y'(v) = \exists v_1.x(v_1) \wedge f(v_1, v)$
`y.f` $=$ `null`	$f'(v_1, v_2) = f(v_1, v_2) \wedge \neg y(v_1)$
`y.f` $=$ `x`	$f'(v_1, v_2) = f(v_1, v_2) \vee (y(v_1) \wedge x(v_2))$

Figure D.1: The predicate-update formulae defining the operational semantics of assignments.

Statement	$y =$ `alloc`(T)
Prepare	$\mathbf{newNode}(\mathbf{addPreds}(S, \{new\}))$
Predicate–update formulae	$isObj'(v) = isObj(v) \vee new(v)$ $y'(v) = new(v)$ $T'(v) = \neg new(v) \wedge T(v) \vee new(v)$
Clean	$\mathbf{removePreds}(S', \{new\})$

Figure D.2: The predicate-update formulae defining the operational semantics of object allocation.

Statement : $iCall_q^{y=p(x_1,\ldots,x_k)}$

Prepare

$\mathbf{clone}(R_{\{x_1,\ldots,x_k\}}(v), \mathbf{addPreds}(S^c, \{new, instance\}))$

Predicate – updateformulae

$$y'(v) = \begin{cases} x_i(v) & : \quad y = h_i \\ 0 & : \quad \text{otherwise} \end{cases}$$

$f'(v_1, v_2) = f(v_1, v_2) \wedge R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2)$

$isObj'(v) = r_{x_1,\ldots,x_k}(v)$

$T'(v) = r_{x_1,\ldots,x_k}(v) \wedge T(v) \vee \exists v_{obj}, instance(v_{obj}, v) \wedge T(v_{obj})$

$isLb'_O(v) = isLb_O(v) \vee new(v)$

$isLb_{CP}{}'(v) = new(v) \wedge$
$\quad \exists v_{obj}.instance(v_{obj}, v) \wedge isCP_{q,\{x_1,\ldots,x_k\}}(v_{obj})$

$lbl'(v_1, v_2) = instance(v_2, v_1) \wedge isCP_{q,\{x_1\ldots x_k\}}(v_2)$

$$\widehat{y}'(v) = new(v) \wedge \begin{cases} \exists v_{obj} : x_i(v_{obj}) \wedge instance(v_{obj}, v) & : \quad y = h_i \\ 0 & : \quad \text{otherwise} \end{cases}$$

$\widehat{f}'(v_1, v_2) = new(v_1) \wedge new(v_2) \wedge$
$\quad \exists v_{obj1}, v_{obj2} : instance(v_{obj1}, v_1) \wedge instance(v_{obj2}, v_2) \wedge$
$\quad\quad f(v_{obj1}, v_{obj2})$

$eq'(v_1, v_2) = (R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2) \wedge eq(v_1, v_2)) \vee$
$\quad (new(v_1) \wedge new(v_2) \wedge$
$\quad\quad \exists v_{obj} : instance(v_{obj}, v_1) \wedge instance(v_{obj}, v_2))$

Clean

Let $S'' = \mathbf{remove}(isObj(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee$
$\qquad\qquad isLb_O(v) \wedge \neg new(v)), S')$
in $\langle U^{S''}, \mathbf{remPreds}(\iota'', \{new, instance\})\rangle$

Figure D.3: The operational semantics for procedure calls in Section 4.8.2: Construction of the structure at the *entry* to a callee. We give the semantics for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q. We assume that p's formal parameters are $h_1, \ldots, h_k$.

Statement : $iRet_q^{y=p(x_1,\ldots,x_k)}$

Prepare

$\mathbf{combine}(\mathbf{addPreds}(S^c, \{inUc, inUx\}),$
$\mathbf{addPreds}(S^x, \{inUc, inUx\}))$

Predicate – updateformulae

$$x'(v) = isObj(v) \wedge \begin{cases} ret(v) & x = y \\ (inUc(v) \wedge x(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v)) \vee & \\ (inUx(v) \wedge \exists v_1 \colon x(v) \wedge inUc(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_1) \wedge & x \in V_q \setminus \{y\} \\ \quad match_{q,\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v)) & \end{cases}$$

$$f'(v_1,v_2) = isObj(v_1) \wedge isObj(v_2) \wedge (inUc(v_1) \wedge inUc(v_2) \wedge f(v_1,v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee inUx(v_1) \wedge inUx(v_2) \wedge f(v_1,v_2) \vee inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{cp} \colon inUc(v_{cp}) \wedge isObj(v_{cp}) \wedge f(v_1,v_{cp}) \wedge match_{q,\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{cp}, v_2))$$

$$isObj'(v) = isObj(v) \wedge (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v))$$

$$T'(v) = T(v) \wedge (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v))$$

$$isLb'_O(v) = isLb_O(v) \wedge inUc(v)$$

$$isLb_{CP}{}'(v) = isLb_{CP}(v) \wedge inUc(v)$$

$$lbl'(v_1,v_2) = isLb_{CP}(v_1) \wedge inUc(v_1) \wedge isObj(v_2) \wedge (inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \wedge lbl(v_1,v_2) \vee inUx(v_2) \wedge \exists v_{cp} \colon R_{\{x_1,\ldots,x_k\}}(v_{cp}) \wedge isObj(v_{cp}) \wedge lbl(v_1,v_{cp}) \wedge match_{q,\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{cp}, v_2))$$

$$\widehat{y}'(v) = \widehat{y}(v) \wedge inUc(v) \wedge isLb_O(v)$$

$$\widehat{f}'(v_1,v_2) = isLb_O(v_1) \wedge isLb_O(v_2) \wedge inUc(v_1) \wedge inUc(v_2) \wedge \widehat{f}'(v_1,v_2)$$

$$eq'(v_1,v_2) = eq(v_1,v_2)$$

Clean

Let $S'' = \mathbf{remove}(isObj(v) \wedge inUc(v) \wedge R_{\{x_1,\ldots,x_k\}}(v) \vee isLb_O(v) \wedge inUx(v), S')$
in $\langle U^{S''}, \mathbf{remPreds}(\iota'', \{inUc, inUx\})\rangle$

Figure D.4: The operational semantics for procedure calls: Construction of the structure at the *return-site* to a caller. We give the semantics for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q. We assume that p's formal parameters are $h_1, \ldots, h_k$.

```
Sll revr(Sll t, Sll r):=
  Sll tn;
  if (t == null) then
    ret = r
  else
    tn = t.n;
    l_d :  t.n = r;
    ret = revr(tn, t);
  fi
```

Figure D.5: A function that *recursively* reverses a list.

D.4 A May-Alias Abstraction of $\mathcal{LSL}$

In this section, we show that Deutsch's abstract-interpretation algorithm [Deu94] can be seen as an abstraction of the $\mathcal{LSL}$ semantics. Also, we provide insight into the clever interprocedural aspects of the analysis.

May-alias algorithms find an upper approximation for the sets of aliased access paths at every program point. The algorithm of [Deu94] is interprocedural, flow-sensitive, and context-sensitive. It handles dynamically allocated memory, recursive functions, and recursive data structures. The algorithm computes (in polynomial time) a (bounded) representation of all the pairs of aliased access paths at every program point.

The algorithm described in [Deu94] is heap modular. The key observation which allows [Deu94] to obtain this property is that a procedure operates uniformly on all aliasing relationships involving variables of pending calls.[1]

$\mathcal{LSL}$ provides insights into the algorithm described in [Deu94]. In particular, the treatment of variables of pending calls, which is one of the most complicated aspects of [Deu94]. For instance, a surprising aspect of the method given in [Deu94] is that recursive procedures are handled in a more precise way than loops. The intuitive reason is that the abstractions of values of variables in the current procedure is different from the abstraction used for values of variables in pending procedures. Specifically, the abstract domain used in [Deu94] is shown to be an abstraction of $\mathcal{LSL}$.

D.4.1 May-Alias Analysis

One of the most intricate aspects of the interprocedural analysis in [Deu94] is the delayed propagation of the effect of destructive updates performed by an invoked function on pending access paths. The algorithm does not represent pending access paths explicitly. Instead, it tracks the effect of the function body on field paths that start at—what we call—cutpoints of the invocation. In particular, it represents (values of) current access paths and (values of) pending access path differently.

This simple observation suffices to see why the analysis of `revr`, a *recursive* function that (destructively) reverses a singly linked list (shown in Figure D.5, originally in [Deu94]) manages to verify that reversing an acyclic list returns an acyclic list, whereas the analysis fails to verify this property for a list-reversal function that uses a loop, e.g., our running example.

The function `revr` reverses a list recursively by invoking itself with the tail (`t`) of the (original) list, which is not reversed yet, and a pointer to the already reversed part (`r`). The analysis handles the destructive update precisely because it can distinguish between the value of `t` in the current call and its values in pending calls by abstracting them differently. However, in the analysis of the loop-based `reverse` function in our running example (where variable `p` plays the same role as `t` in Figure D.5) , the analysis cannot distinguish between the value of `p` in the different iterations. Note that this loss of information is inherent in the may-alias analysis. In particular, it does not depend on the algorithm that abstracts the access paths.

[1] The method of [Deu94] applies to programs with cutpoints. However, the lack of *must*-alias information may lead to a loss of precision in the analysis of destructive updates.

D.4.2 A Galois Connection between $\mathcal{LSL}$ and Deutsch's May-Alias Abstraction

In this section we define a Galois connection between sets of $\mathcal{LSL}$ states and the abstract domain of [Deu94] for may-alias analysis.

The algorithm of [Deu94] computes (in polynomial time) at every program point l a set of *symbolic alias pairs* (*SAP*s). The computed set represents (in a bounded way) any pair of *current* access paths that are aliased at l. As explained in Section D.4.1, The algorithm does not represent pending access paths explicitly. Instead, at the call-site, the algorithm *generalizes* any *SAP* representing an alias with an access path that starts at an actual parameter. The generalized *SAP* contains: (i) a representation of the access path that starts at the actual parameter, where the root of the access path (i.e., the actual parameter) is substituted by its corresponding formal parameter, and (ii) a name of a *generic object*. A generic object represent—what we call—a cutpoint of the invocation. The name of the generic object is determined uniquely by the access path it is aliased with. We denote by *APG* the set of access paths enriched with generic object names (i.e., *APG* contains access paths that start at a variable or a generic object name.).

In our terminology, generic objects are an abstraction of the cutpoints of the invocation, and the name of the generic object is an abstraction of the cutpoint-label based on its content. The use of generic object names in the analysis of a return statement is an approximation of the way cutpoint-labels are used in $\mathcal{LSL}$.

The actual representation of symbolic alias pairs (*SAP*s) is immaterial for the definition of the Galois connection. All we rely on is that the set $UR = 2^{SAP}$ of all symbolic alias relation forms a lattice ordered by $\sqsubseteq_{SAP}$ and equipped with a join operator $\sqcup_{SAP}$.[2] We make use of the function $Factor : AccPath \times AccPath \to SAP$, defined in [Deu94], which maps a pair of unbounded (aliased) access paths, possibly starting with a generic object name, to its most precise representation by a *SAP*. We also make use of the function $makeGenericName : AccPath \to APG$, also defined in [Deu94], which maps an access path that starts with a formal parameter to the generic object name it determines.

To establish the Galois connection between the set of program states (ordered by set inclusion) and *UR*, it suffices to show a *representation function* that maps a program state to its "most precise representation" in *UR* (e.g., see [NNH99]). The function $\beta^p_{may} \colon \Sigma^p_L \to SAP$, defined in Figure D.6 is a representation function. It is parameterized for every function p in the program by the set of the p's local variables (V_p) and formal parameters (F_p).

The function β^p_{may} is defined as a composition of two functions: (i) $toPairs^p \colon \Sigma^p_L \to 2^{APG \times APG}$, which maps a program state of function p, $\sigma^p_L \in \Sigma^p_L$, to pairs of (unbounded) access paths enriched with object names; and (ii) $boundPairs \colon 2^{APG \times APG} \to UR$, which bounds the resulting set by mapping it to a (bounded) set of symbolic access pairs.

The function *toPairs* converts a program state to a (bounded) alias relation in two steps: (i) it creates the equivalence relation (*AP*) by pairing any two generalized access paths that belong to the same equivalence class; (ii) it "recovers" the generic object names out of any generalized access path that starts with a cutpoint or a formal parameter by invoking *generic*. The special treatment for formal parameters is required because [Deu94] considers objects pointed-to by actual parameters as (trivial) cutpoints, where we do not. A bounded representation is achieved by applying *Factor* pointwise and taking the least upper bound of the resulting set of symbolic access paths.[3]

[2] In [Deu94], The set *UR* is actually parameterized by the numeric lattice used in the analysis. Since the parameterization is not relevant for our purposes, we ignore this issue.

[3] In [Deu94], special care needs to be taken in case the analysis is parameterized by a lattice with infinite chains. In particular, $\sqsubseteq_{SAP}$ is not necessarily bounded. For simplicity, we assume this is not the case.

β^p_{may}: $\Sigma^p_L \to UR$ s.t.
$\beta^p_{may} = boundPairs \circ toPairs^p$

$toPairs^p$: $\Sigma^p_L \to 2^{APG \times APG}$ s.t.
$toPairs^p(\langle CPL, A\rangle) =$ Let
$AP = \{\langle \alpha, \beta\rangle \mid \exists a \in A,$ s.t. $\alpha \in a$ and $\beta \in a\}$
in $\bigcup_{\langle \alpha,\beta\rangle \in AP} \left\{ \langle \alpha', \beta'\rangle \;\middle|\; \begin{array}{l} \alpha' \in generic^p(CPL, \alpha), \\ \beta' \in generic^p(CPL, \beta) \end{array} \right\}$
Where
$generic^p$: $2^{CPL} \times GAccPath \to 2^{APG}$ s.t.
$generic^p(CPL, \langle r, \delta\rangle) =$

$$\begin{cases} \{\langle r, \delta\rangle\} & r \in V_p \setminus F_p \\ \{\langle r, \delta\rangle, \langle makeGenericName(\langle r, \epsilon\rangle), \delta\rangle\} & r \in F_p \\ \{\langle makeGenericName(\alpha), \delta\rangle \mid \alpha \in r\} & r \in CPL \end{cases}$$

$boundPairs$: $2^{APG \times APG} \to UR$ s.t.
$boundPairs(AliasRel) =$
$\bigsqcup_{SAP}\{Factor(\langle \alpha, \beta\rangle) \mid \langle \alpha, \beta\rangle \in AliasRel\}$

Figure D.6: β^p_{may} is a representation function that maps a memory state of function p to its most precise representation as sets of symbolic access path.

Appendix E

Appendix for Chapter 5

This chapter provides formal details pertaining to Chapter 5. Specifically, the following appendixes provides formal details which were omitted from the body of Chapter 5:

- Section E.1 defines our technique for handling procedure calls using a stack of program states, and exemplifies it by defining our version of the standard store-based semantics for pointer programs. It also defines the notion of interprocedural execution traces.
- Section E.2 provides the formal details which were omitted in Sections 5.5 and 5.6.

E.1 Interprocedural Lifting Semantics

In this section, we present a technique which allows to *lift* a concrete intraprocedural semantics into an interprocedural semantics. Specifically, we suggest a way to extend a semantics which supports only atomic (intraprocedural) statements to handle procedure invocations.

The main idea is to replace the representation of the program state used by the intraprocedural semantics by a *stack* of program states. A standard stack of activation records, contains only the local variables, the current program points (program counter), the return addresses. In contrast, the stack that we use stores *whole* program states.

Our technique requires to extend the intraprocedural semantics with operations that defines the memory state of the invoked procedure when its execution starts (according to the state of the caller at the call-site) and the memory state of the caller when it regains control (according to the caller's memory state at the call-site and the callee's memory state at the exit-site). These operations are similar to the ones used in a large-step semantics [Kah87] to define the memory state on which the body of an invoked procedure is executed (entry memory state) and the memory state resulting after a procedure call (return memory state), see, e.g., Chapters 3 and 4.

The ideas in this section are heavily influenced by the formulation of an *interprocedural analysis* using a stack of *abstract* memory states in [KS92].

E.1.1 Procedure Representation by Flow Graphs

In the following, we assume that the bodies of procedures are represented in a standard way by their *flow graphs*. A flow graph of a procedure p is a rooted directed graph $G_p = \langle N_p, E_p, s_p, e_p \rangle$.

G_p's nodes, $N_p \subset \mathcal{PP}$, are *program points*. G_p is rooted at s_p, the *entry-site* to p. The program point e_p is p's *exit-site*. Every node, except s_p (resp. e_p), is the target (resp. source) of an edge $e \in E_p$. For simplicity, we assume that the sets of program points of different procedures are disjoint. G_p's edges, $E_p \subseteq N_p \times N_p$, are associated with *atomic* statements and *procedure calls*.

The function $stmt_{G_p}(e)$ maps flow graph G_p edges to statements. The function out_{G_p} maps a given program point in N_p to the set of its successors[1], i.e., $out_{G_p}(n) = \{n' \in N_p \mid \langle n, n' \rangle \in E_p\}$. (We omit the G_p subscript when it is clear from the context). For simplicity, we assume that the edges emanating from s_p, as well as the ones entering into e_p, are associated with nop statements. When an edge $e = \langle n_c, n_r \rangle$ is associated with a procedure

[1] The intended meaning of several edges emanating from a single program point is that a successor is chosen non deterministically.

call, we say that n_c is a *call-site* and that n_r is its *corresponding return-site*. We assume every call-site n_c has exactly one return-site which we denote by $return(n_c)$. Similarly, we assume every return-site n_r has exactly one call-site, denoted by $call(n_r)$.

The function $fg(n)$ maps a program point to the (unique) flow graph which contains n. The function $fg(p)$ maps a procedure identifier p to p' flow graph. The function $proc(G)$ maps a flow graph G of procedure p to p's procedure identifer.

A program is comprised of a set of procedures, including a distinguished `main` procedure. We denote the set of all procedures in a program P by $procs(P)$, and the set of all program points in P by $PP(P)$, i.e., $PP(P) = \bigcup_{p \in procs(P)} N_p$, where N_p are the program points of procedure p.

E.1.2 Intraprocedural Semantics

An *intraprocedural semantics S manipulating memory states $\sigma \in \Sigma$* defines a *meaning* for every intraprocedural statement st as a binary relation over a set of memory states $[\![st]\!]_S \subseteq \Sigma \times \Sigma$. A pair of memory states $\langle \sigma, \sigma' \rangle \in [\![st]\!]_S$ (also denoted by $\sigma' \in [\![st]\!]_S(\sigma')$, see Definition A.1.4) iff the execution of st in memory state σ may lead to memory state σ'.

Definition E.1.1 (Program states) *A **program state** of a program P according to a semantics which manipulates memory states Σ is a pair comprised of a program point and a memory state, $\langle pp, \sigma \rangle \in PP(P) \times \Sigma$.*

An intraprocedural semantics S associates to every (single-procedure) program P a *transition system* between program states $tr_P \subseteq (PP(P) \times \Sigma) \times (PP(P) \times \Sigma)$. We write $\langle pp, \sigma \rangle \overset{tr_P}{\rightsquigarrow} \langle pp', \sigma' \rangle$ for $\langle \langle pp, \sigma \rangle, \langle pp', \sigma' \rangle \rangle \in \overset{tr_P}{\rightsquigarrow}$. A transition $\langle pp, \sigma \rangle \overset{tr_P}{\rightsquigarrow} \langle pp', \sigma' \rangle$ indicates that (i) there is an edge $\langle pp, pp' \rangle \in E$, i.e., $pp' \in out(pp)$, and (ii) the execution of a statement $st = st(\langle pp, pp' \rangle)$ in memory state σ may lead to memory state σ', i.e., $\sigma' \in [\![st]\!]_S(\sigma)$.

E.1.3 Interprocedural Lifting

We lift an intraprocedural semantics, as defined above, to an interprocedural semantics which is capable of handling procedure calls. The main idea is that the interprocedural semantics associates with every program P a transition system between *stacks of program states* (instead of a transition system between program states).

Intuitively, the interprocedural semantics maintains a stack of program states. The program state of the *current* (active) procedure is stored at the top of the stack. Intraprocedural statements have access only to the top of the stack.

Interprocedural statements make use of the (unbounded number of) program points stored in the stack to ensure that a procedure which was invoked at a call-site n_c returns to the corresponding return-site, $return(n_c)$.

The memory states stored in the stack are used to record the memory states of the caller as it was when the control reached the invocation call-site. Our lifting technique does not dictate the memory states at the entry-sites and return-sites. Instead, it requires two operations, $Call$ and Ret, which specify, for every procedure call statement $y = p(x_1, \ldots, x_k)$, the effect of transferring control from the caller to the callee, and vice versa:

$$\begin{array}{rcl} [\![Call_{y=p(x_1,\ldots,x_k)}]\!] & \subseteq & \Sigma \times \Sigma \\ [\![Ret_{y=p(x_1,\ldots,x_k)}]\!] & \subseteq & (\Sigma \times \Sigma) \times \Sigma \end{array}$$

Intuitively, the semantics utilizes the above relations to handle procedure calls as follows:

- When a procedure is invoked, the semantics *pushes* a new program state, which we refer to as the *entry state*, to the top of the stack. The entry state is comprised of the callee's entry-site and an *entry memory state*. The entry memory state depends on the memory state at the call-site and is specified by the meaning of a *Call* operation.
- Execution of the statements in the callee's body is continued as in the case of the intraprocedural statements, manipulating the program state at the top of the stack.
- When a procedure execution reaches the exit-site, the semantics pops the callee's program state from the top of the stack. The caller's execution is continued from the return-site corresponding to the call-site stored in the stack. The memory state at the return site is updated by the Ret relation according to the call memory state and the exit memory state.

$$
\begin{aligned}
&newstack : \mathcal{PP} \times \Sigma \to \mathcal{STK}_\Sigma \\
&top : \mathcal{STK}_\Sigma \to \mathcal{PP} \times \Sigma \\
&push : \mathcal{STK}_\Sigma \times \mathcal{PP} \times \Sigma \to \mathcal{STK}_\Sigma \\
&pop : \mathcal{STK}_\Sigma \hookrightarrow \mathcal{STK}_\Sigma \\
&|\cdot| : \mathcal{STK}_\Sigma \hookrightarrow \mathbb{N} \\
&top(newstack(pp, \sigma)) = \langle pp, \sigma \rangle \\
&top(push(stk, pp, \sigma)) = \langle pp, \sigma \rangle \\
&pop(push(stk, pp, \sigma)) = stk \\
&|stk| = \begin{cases} 1 & stk = newstack(pp, \sigma) \\ 1 + |pop(stk)| & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure E.1: Axiomatic definition of a stack of program states. Note that $pop(newstack(pp, \sigma))$ is undefined.

Formally, a (lifted) interprocedural semantics which is based on an intraprocedural semantics manipulating memory states $\sigma \in \Sigma$, manipulates *stacks* of program states, $stk \in \mathcal{STK}_\Sigma$. The equational definition of the set $\mathcal{STK}_\Sigma$ in Figure E.1 provides the following stack-manipulating operations: - $newstack$ creates a new stack containing a single program state; - $push$ pushes a program state to the top of the stack; - top retrieves the program state from the top of the stack; - pop pops the program state from the top of the stack; and - $|\cdot|$ returns the number of elements in the stack.

The use of stacks of program states allows us to formalize the notion of the *current program state*.

Definition E.1.2 (Current program state) *The **current program state of a stack** $stk \in \mathcal{STK}_\Sigma$ is the program state $\langle pp, \sigma \rangle = top(\sigma)$. The memory state σ, denoted by $cur_{state}(stk)$, is the **current memory state of stack** stk. The program point pp, denoted by $cur_{pc}(stk)$, is the **current program point of stack** stk. The procedure $proc(fg(pp))$, denoted by $cur_{proc}(stk)$, is the **current procedure of stack** stk.*

A (lifted) interprocedural semantics $\mathcal{S}$ associates with every program P a transition system $str \subseteq \mathcal{STK}_\Sigma \times \mathcal{STK}_\Sigma$ between *stacks* of program states. The transition system, defined in defined in Figure E.2, is parameterized by the intraprocedural transition relation, tr_P, and the meaning of $Call.$ and $Ret.$ operations. We write $stk \overset{str_P}{\rightsquigarrow} stk'$ for $\langle stk, stk' \rangle \in \overset{str_P}{\rightsquigarrow}$.

A transition $stk \overset{str_P}{\rightsquigarrow} stk'$ indicates that one of the following holds:

- $stk \overset{str_P^{Intra}}{\rightsquigarrow} stk'$: stk' results from an application of an intraprocedural statement on the current memory state of stk.
- $stk \overset{str_P^{Call}}{\rightsquigarrow} stk'$: The current program point in stk is a call-site. The entry-state which results from applying a $Call$ statement to the current memory state of stk, is pushed into stk, resulting in stk'.
- $stk \overset{str_P^{Ret}}{\rightsquigarrow} stk'$: The current program point in stk is an exit-site. The return-state which results from applying a Ret statement on the two topmost memory states in stk, the call memory state and the exit memory state, is pushed into stk, after the call state and the exit state have been popped, resulting in stk'.

The following definition formalizes the notion of the statement executed in a transition.

Definition E.1.3 (Executed statement) *The **executed statement in a transition** $stk \overset{str_P}{\rightsquigarrow} stk'$, denoted by $stmt(\langle pp, pp' \rangle)$, where $pp = cur_{pc}(stk)$ and $pp' = cur_{pc}(stk')$, is:*

- *$st_{fg(pp)}(\langle pp, pp' \rangle)$ if program points pp and pp' belong to the same procedure, i.e., $fg(pp) = fg(pp')$.*
- *$Call_{y=p(x_1,\ldots,x_k)}$ if pp is a call-site, $st_{fg(pp)}(\langle pp, return(pp) \rangle) \equiv y = p(x_1, \ldots, x_k)$, and pp' is the entry-site of p.*
- *$Ret_{y=p(x_1,\ldots,x_k)}$ if pp' is a return-site, $st_{fg(pp')}(\langle call(pp'), pp' \rangle) \equiv y = p(x_1, \ldots, x_k)$, and pp is the exit-site of p.*

$$
\begin{array}{l}
str_P \subseteq \mathcal{STK}_\Sigma \times \mathcal{STK}_\Sigma \;\; s.t. \\
str_P = str_{Intra} \cup str_{Call} \cup str_{Ret} \\
str_P^{Intra} = \\
\left\{ \langle stk, stk' \rangle \;\middle|\; \begin{array}{c} \langle pp, \sigma \rangle = top(stk), \\ \langle pp, \sigma \rangle \overset{tr_P}{\rightsquigarrow} \langle pp', \sigma' \rangle, \\ stk' = push(pop(stk), pp', \sigma') \end{array} \right\} \\
str_P^{Call} = \\
\left\{ \langle stk, stk' \rangle \;\middle|\; \begin{array}{c} \langle pp, \sigma \rangle = top(stk), \\ pp_r = return(pp), \\ st_{fg(pp)}(\langle pp, pp_r \rangle) \equiv y = p(x_1, \ldots, x_k), \\ \sigma_e \in [\![Call_{y=p(x_1,\ldots,x_k)}]\!](\sigma), \\ stk' = push(stk, s_p, \sigma_e) \end{array} \right\} \\
str_P^{Ret} = \\
\left\{ \langle stk, stk' \rangle \;\middle|\; \begin{array}{c} \langle e_p, \sigma \rangle = top(stk), \\ \langle pp_c, \sigma_c \rangle = top(pop(stk)), \\ pp_r = return(pp_c), \\ st_{fg(pp_c)}(\langle pp_c, pp_r \rangle) \equiv y = p(x_1, \ldots, x_k), \\ \sigma_r \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!](\langle \sigma_c, \sigma \rangle), \\ stk' = push(pop(pop(stk)), pp_r, \sigma_r) \end{array} \right\}
\end{array}
$$

Figure E.2: Lifted interprocedural transition system for an arbitrary program P. p is an arbitrary procedure; $fg(p) = \langle N_p, E_p, s_p, n_p \rangle$. Σ is the set of manipulated memory states.

E.1.4 Interprocedural Paths, Traces, and Executions

In this section, we define the notion of the *reachable interprocedural traces of a program*. Based on this notion, we define the notions of *reachable memory states* and the *meaning of a procedure*. The latter is defined as an input-output relation. According to our definition, a procedure may have different meanings in different programs. However, the difference amounts to different sets of input memory states. This indicates that it is possible to define a functional meaning for a procedure as a transformer from *input states* to *output states* in a program independent manner.

In the rest of the section, we assume that P is an arbitrary program and that $\mathcal{S}$ is an arbitrary (lifted) interprocedural semantics manipulating memory states $\sigma \in \Sigma$. We denote by str_P the transition relation associated by $\mathcal{S}$ to P. We assume that the execution of every program begins at a (designated) *initial memory state*, $\sigma_0 \in \Sigma$.

Definition E.1.4 (Executions) *A sequence $\pi \in \Pi_{\mathcal{STK}_\Sigma}$ is an **execution** of program P if $\langle \pi(i), \pi(i+1) \rangle \in str_P$ for every $1 \leq i < |\pi|$.*

Definition E.1.5 (Paths) *The **path induced by a program trace** π, denoted by $path(\pi)$, is a sequence of program points $\overline{pp}$ such that $\overline{pc}(i) = cur_{pc}(\pi(i))$.*

Definition E.1.6 (Initial and final memory states) *The **initial resp. final memory state of an execution** π, denoted by $in(\pi)$ resp. $out(\pi)$, is the current memory state of $top(\pi(1))$ resp. $top(\pi(|\pi|))$.*

Definition E.1.7 (Feasible executions) *An execution trace π of program P is **feasible** if $\pi(0) = newstack(s_{main}, \sigma_0)$. We denote the set of feasible program execution traces of program P according to semantics $\mathcal{S}$ by $\Pi_{\mathcal{S}}^P$.*

Definition E.1.8 (Reachable memory states) *Let pp be a program point in program P, $pp \in PC(P)$. A memory state $\sigma \in \Sigma$ is a **reachable memory state at** pp **(according to semantics $\mathcal{S}$)** if there exists a feasible execution trace $\pi \in \Pi_{\mathcal{S}}^P$ such that pp is the current program point of $\pi(|\pi|)$ and σ is the current memory state of $\pi(|\pi|)$.*

We denote the set of reachable memory states at pp in program P (according to semantics $\mathcal{S}$) by $\mathcal{R}(pp)_{\mathcal{S}}^P$.

Example E.1.9 We exemplify the lifting of intraprocedural semantics to the interprocedural setting by lifting $\mathcal{GSB}$, the standard intraprocedural store-based semantics for heap-manipulating programs, defined in Section 2.2.

The meaning of every intraprocedural statement $st \in Stmt$ in $\mathcal{GSB}$ is defined in Figure 2.4 by means of a transition relation $\overset{GSB}{\rightsquigarrow} \subseteq \mathcal{S}_G \times Stmt \times \mathcal{S}_G$. Below, we extend the $\overset{GSB}{\rightsquigarrow}$ relation to describe the meaning of $Call$ and $Return$ statements pertaining to any arbitrary procedure call $y = p(x_1, \ldots, x_k)$:

$$\langle Call_{y=p(x_1,\ldots,x_k)}, s^c_G \rangle \overset{GSB}{\rightsquigarrow} \langle L_c, [z_i \mapsto \rho_c(x_i) \mid 1 \leq i \leq k], h_c \rangle$$
$$\langle Ret_{y=p(x_1,\ldots,x_k)}, s^c_G, s^x_G \rangle \overset{GSB}{\rightsquigarrow} \langle L_x, \rho_c[y \mapsto \rho_x(ret)], h_x \rangle$$
where $s^c_G = \langle L_c, \rho_c, h_c \rangle$ and $s^x_G = \langle L_x, \rho_x, h_x \rangle$

The $Call$ operation initializes the formal parameters using the values of the actual parameters, and initiates the invoked procedure execution in a memory state that contains the caller's heap. The Ret operation copies into the return memory state the callee's heap at the exit-site, and restore the values of the caller's variables from the call memory state (stored in the stack), except for y, which is assigned the callee's return value.

Note that in this example, we lift a global-heap semantics. Thus, the semantics treats the whole heap as a single global resource which is "passed" as a whole in every procedure call. In particular, as a result, the heap part of the memory state in the call-site is not needed to define the memory state at the return-site.

E.2 Formal Details Pertaining to the $\mathcal{DOS}$ Semantics

Section E.2.1 provides the formal (and rather standard) definitions of *reachability* and *connectivity* in the context of Chapter 5. Section E.2.2 formalizes the notion of implicit components and implicit component graphs.

E.2.1 Reachability, Connectivity, and Domination

In this section, we give formal definitions for the notions of *reachability* and (undirected) *connectivity* in $\mathcal{DOS}$ memory states. We also formalize the notion of domination. These definitions are based on the corresponding standard notions in 2-level stores. Intuitively, location l_2 is *reachable from* (resp. *connected to*) a location l_1 in a memory state σ if there is a directed (resp. undirected) path in the heap of σ from l_1 to l_2. A locations l is *reachable* in σ if it is reachable from a location which is pointed to by some variable. An object l is a *dominator* if every access path pointing to an object reachable from l, must traverse through l. Note that the inaccessible value is treated, basically, as a *null* value, *i.e.*, it cannot lead to an object.

Definition E.2.1 (Heap path) *A sequence π of location is a **directed heap path** in a heap $h \in \mathcal{H}$, if for every $0 \leq i < |\pi| - 1$ there exists $f_i \in \mathcal{F}$ such that $h(\pi(i), f_i) = \pi(i+1)$. A directed heap path π **goes from** l_1, if $\pi(0) = l_1$, it **goes to** l_2 if $\pi(|\pi| - 1) = l_2$.*

*A sequence π of location is an **undirected heap path** in $h \in \mathcal{H}$, if for every $0 \leq i < |\pi| - 1$ there exists $f_i \in \mathcal{F}$ such that either $h(\pi(i), f_i) = \pi(i+1)$ or $h(\pi(i+1), f_i) = \pi(i)$. A directed heap path is π **connecting** l_1 and l_2, if $\pi(0) = l_1$ and $\pi(|\pi| - 1) = l_2$, or vice versa.*

*A heap path π **traverses through** l if there exists $i,\ 1 \leq i < |\pi|$ such that $l = \pi(i)$.*

Definition E.2.2 (Reachability) *A location l_2 is **reachable from** a location l_1 in a memory state $\sigma = \langle \rho, L, h, t, m \rangle$, if there is a directed heap path in h going from l_1 to l_2.*

Definition E.2.3 (Reachable locations) *Location l is **reachable** in σ if it is reachable from a location which is pointed to by some variable. We denote the set of **reachable locations** in $\sigma \in \Sigma_D$ by $\mathcal{R}(\sigma)$, i.e., $\mathcal{R}(\sigma) = \{l \in L | x \in \mathcal{V} \text{ and } l \text{ is reachable in } \sigma \text{ from } \rho(x) \in Loc\}$.*

Definition E.2.4 (Connectivity) *Locations l_1 and l_2 are **connected** in a memory state $\sigma = \langle \rho, L, h, t, m \rangle$, if there is an* undirected *path in h connecting l_1 to l_2.*

Definition E.2.5 (Domination) *A set of locations $D \subseteq Loc$ are **dominator** (or **dominate their reachable heap**) in memory state $\sigma = \langle \rho, L, h, t, m \rangle \in \Sigma_D$, if for every $x \in \mathcal{V}$ such that $l_x = \rho(x) \in Loc$, every directed heap path in h* from l_x to *a location which is reachable from a location in D, traverses through a location in D.*

E.2.2 Components

We formalize the notion of components, implicit components decomposition using the definitions of *reachability* given in Appendix E.2.1. We use the auxiliary function $succ_h(L) = \{h(l)f \in Loc \mid l \in L, f \in \mathcal{F}\}$. Function $succ_h(L)$ computes the set of objects which, in heap h, are pointed to by a field of an object in L.

Definition E.2.6 (Components) *The* domain of components *is $\mathcal{C} = 2^{Loc} \times 2^{Loc} \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M}$. A* component *$c = \langle I, L, R, h, t, m \rangle \in \mathcal{C}$ is a 6-tuple. L contains c's internal objects; $I \subseteq L$ and $R \subseteq Loc \setminus L$ constitute c's spatial interface. The heap $h \in L \hookrightarrow \mathcal{F} \hookrightarrow (L \cup R \cup \{null, \ominus\})$ defines the values of fields for objects inside c. The type map $t \in (L \cup R) \to \mathcal{T}$ defines the types of the objects inside c and in its rim. m is c's* component module. *We say that component c belongs to m. For every $l \in L, m(t(l)) = m$. For every $l \in R, m(t(l)) \neq m$.*

A component can be in one of two states: sealed *or* unsealed. *In the context of a memory state, we are always able to tell the state of a component (see below). Thus, formally, we only place an additional restriction on sealed components: If c is a sealed component then $R = \{h(l)f \in Loc \mid l \in L, f \in \mathcal{F}\}$.*

The types of the reachable objects in a memory state σ induce a (unique) *implicit component decomposition* of σ: The current component contains all the reachable locations that belong to the current module. Every sealed component contains a maximal set of reachable $\mathcal{M}$-connected locations. (These sets are mutually disjoint by definition).

Definition E.2.7 ($\mathcal{M}$-Connectivity) *Locations l_1 and l_2 are **$\mathcal{M}$-connected** in a memory state $\sigma = \langle \rho, L, h, t, m\rangle$, denoted by $l_1 \overset{\mathcal{M}}{\rightsquigarrow}_\sigma l_2$, if there is an undirected heap path π connecting l_1 and l_2 such that for every $0 \leq i < |pi|$, $m_\sigma(l_1) = m_\sigma(\pi(i))$.*

The internal structure of an implicit component is induced by a *restriction* of a $\mathcal{DOS}$ memory state σ on a set of *reachable* locations who belong to the same module. Note that references from unreachable locations do not count. Also note that because our semantics is cutpoint-free, *unreachable locations in the local-heap are also unreachable in the global-heap*.

Definition E.2.8 (Implicit components) *A sealed component $c \in \mathcal{C}$ is an **implicit sealed component** of a $\mathcal{DOS}$ memory state $\sigma = \langle \rho, L, h, t, m\rangle$ if there exists a module $m^c \neq m$ and a nonempty set $L^c \subseteq \mathcal{R}(\sigma)$ of reachable objects such that for every $l_1 \in L^c$ and $l_2 \in \mathcal{R}(\sigma)$, if $l_1 \overset{\mathcal{M}}{\rightsquigarrow}_\sigma l_2$, then $l_2 \in L^c$ and $c = \langle I, L^c, R, h|_{L^c}, t|_{L^c \cup R}, m^c\rangle$ where $I = \{l \in L^c \mid l \in succ_h(\mathcal{R}(\sigma) \setminus L^c)\} \cup \{\rho(x) \in L^c \mid x \in \mathcal{V}\}$ and $R = succ_h(L^c) \setminus L^c$. (Note that, in particular, $m_\sigma(l_1) = m_\sigma(l_2) = m_c$.)*

*The **implicit current component** of σ is an unsealed component $c \in \mathcal{C}$ such that $c = \langle I, L^*, R, h|_{L^*}, t|_{L^* \cup R}, m\rangle$ where $L^* = \{l \in \mathcal{R}(\sigma) \mid m_\sigma(l) = m\}$, $I = \{\rho(x) \in L^* \mid x \in \mathcal{V}\}$, and $R = (succ_h(L^*) \cup \{\rho(x) \in Loc \mid x \in \mathcal{V}\}) \setminus L^*$.*

*We denote the **set of sealed components in a memory state** $\sigma \in \Sigma_D$ by $\mathcal{C}(\sigma)$. The implicit current component of σ is denoted by $c^\star(\sigma)$.*

Note that the entry locations of the unsealed current component are determined only by the values of variables. The acyclicity of the module dependency relation ensures that (in the local-heap) there are no references from objects inside sealed components to objects inside the current component.

The component decomposition of a memory state σ naturally induces its *implicit component graph*, which is a directed graph whose nodes are the implicit components of σ and its edges reflect the inter-component reference structure.

Definition E.2.9 (Induced component graphs) *The* induced component graph *of a memory state σ denoted by $\mathcal{CG}(\sigma)$, is a directed graph $\mathcal{CG}(\sigma) = \langle \mathcal{C}(\sigma), E\rangle$ such that $E \subseteq \mathcal{C}(\sigma) \times \mathcal{C}(\sigma)$ and $\langle c_1, c_2\rangle \in E$ iff $R_1 \cap I_2 \neq \emptyset$, where $c_1 = \langle I_1, L_1, R_1, h_1, t_1, m_1\rangle$ and $c_2 = \langle I_2, L_2, R_2, h_2, t_2, m_2\rangle$.*

A component graph is ensured to be connected:

- the target of a reference between two objects located inside different implicit components is also in the rim of the component containing the reference's source.
- References from a local variable to a location outside the current module are treated as an inter-component reference leaving the current component. In particular, all such entry locations are also in the rim of the implicit current component.

Appendix F

Interprocedural Tabulation Algorithm

In this chapter, we describe the iterative interprocedural local-heap shape-analysis algorithm. For simplicity, we describe it in a generic way as an algorithm that manipulates shape-graphs. In this thesis, shape-graphs are implemented by *3-valued* logical structures.

The algorithm computes procedure summaries by tabulating input shape-graphs to output shape-graphs. The tabulation is restricted to shape-graphs that *occur in the analyzed program*. The abstract domain is the powerset of *shape-graphs* (2^{SG}) with set-union as the *join* operator. The abstract-transformers are always applied point-wise, thus they distribute over the join operator (e.g., see [NNH99]). The algorithm remains sound in case the join operator is an over-approximation of set union.

The algorithm is a variant of the IFDS-framework [RHS95] adapted to work with local-heaps. The main difference between our framework and [RHS95] is in the way return statements are handled: In [RHS95], the dataflow facts that reach a return-site come either from the call-site (for information pertaining to local variables) or from the exit-site (for information to pertaining global variables). In our case, the information about the heap is obtained by *combining* pair-wise the shape-graphs at the call-site with the shape-graphs at the exit-site: the information about the values of local variables and fields of objects that point to the part of the heap which was *not* passed to the callee is passed as-is from the call-site. The information about the values of fields of objects in the part of the heap which was passed to the callee is taken as-is from the exit-site. The information about the value of the caller's local variables and the values of fields of objects that were not passed to the callee, but point to objects that are passed to the callee, are computed by the *combine* operation (see Section F.2).

F.1 Program Model

We represent a program P in a standard manner by the set of *control-flow-graphs* of its procedures (with a distinguished `main` procedure), connected by a set of interprocedural call/return edges. The control-flow-graph CFG_p of a procedure p, is comprised of a set of nodes N_p, representing program locations, and a set of intraprocedural edges $E_p \subseteq N_p \times N_p$ labeled with program statements. We assume that every CFG_p has a single entry-node, $entryp$, and a single exit-node, $exitp$.

We partition the set $N^\star$ of all CFG nodes in the program into five subsets: $Entry^\star$, $Exit^\star$, $Call^\star$, $Ret^\star$, and $Intra^\star$, corresponding to the sets of all entry-nodes, exit-nodes, call-sites, return-sites, and all other nodes, respectively.

The procedural control-flow graphs are connected by a set of interprocedural edges $E_{inter} \subseteq Call^\star \times Entry^\star \cup Exit^\star \times Ret^\star$. We denote the set of all program edges by $E^\star = \bigcup_{p \in pgm} E_p \cup E_{inter}$.

For simplicity, we guarantee that return-sites are not call-sites or exit-sites, by augmenting each return-site with a single `nop` operation.

In the sequel we denote the set of outgoing edges for a node $n \in N^\star$ by $out(n)$, and the statement that labels an edge $\langle n, n' \rangle \in E^\star$ by $stmt(\langle n, n' \rangle)$. We also use *callee*$(n_{call})$ and $return(n_{call})$ to denote the target of the call at n_{call} and the return-site of n_{call}, respectively. For an entry-node $n \in Entry^\star$, we denote the matching exit-node by $exitn$.

```
proc tabulate(Program P, SG sg_0)
  worklist = {⟨entrymain : ⟨sg_0, sg_0⟩⟩}
  while (worklist ≠ ∅)
    remove an event ⟨n : ⟨sg_entry, sg⟩⟩ from worklist
    if n ∈ Entry* ∪ Ret* ∪ Intra* then
      foreach ⟨n, n'⟩ ∈ out(n)
        foreach sg' ∈ apply(stmt(⟨n, n'⟩), sg)
          if ⟨sg_entry, sg'⟩ ∉ PathSet(n') then
            propagate(n', ⟨sg_entry, sg'⟩)
    else if n ∈ Call*
      n_callee^entry = callee(n)
      if not applicable(stmt(⟨n, n_callee^entry⟩), sg) then
        HALT
      foreach sg' ∈ extract(stmt(⟨n, n_callee^entry⟩), sg)
        add ⟨n, ⟨sg_entry, sg⟩⟩ to CTXs(n_callee^entry) sg'
        if ⟨sg', sg'⟩ ∉ PathSet(n_callee^entry) then
          propagate(n_callee^entry, ⟨sg', sg'⟩)
        else
          n_exit = exit n_callee^entry
          foreach sg_exit ∈ sm(n_callee^entry)sg'
            addToRet(n_exit, sg_exit, return(n), ⟨sg_entry, sg⟩)
    else //n ∈ Exit*
      foreach⟨n_call, ⟨sg_e, sg_c⟩⟩ ∈ CTXs(n_callee^entry) sg_entry
        addToRet(n, sg, return(n_call), ⟨sg_entry, sg_call⟩)

proc addToRet(N_callee n_exit, SG sg_x, N_caller n_ret, SG × SG ⟨sg_e, sg_c⟩)
  foreach sg' ∈ combine(stmt(⟨n_exit, n_ret⟩), ⟨sg_c, sg_x⟩,)
    if (⟨sg_e, sg'⟩ ∉ PathSet(n_ret)) then
      propagate(n_ret, ⟨sg_e, sg'⟩)
```

Figure F.1: The tabulation algorithm.

F.2 Tabulation Algorithm

We describe the algorithm using the following operations as "black boxes":

- *apply* : $Stmt \times SG \rightarrow 2^{SG}$ applies the abstract transformer associated with a given *intraprocedural* statement to a given shape-graph and returns the resulting set of shape-graphs.
- *applicable* : $Stmt \times SG \rightarrow \{$**true**, **false**$\}$ verifies that the given procedure call statement can be applied to the given shape-graph.
- *extract* : $Stmt \times SG \rightarrow 2^{SG}$ applies the abstract transformer associated with the given call-statement to the given shape-graph. Thus, computing the shape-graph that represents the local-heap which is passed to the callee.
- *combine* : $Stmt \times SG \times SG \rightarrow 2^{SG}$ computes the shape-graph representing the local-heap of the caller at the return-site by applying the associated abstract transformer when control returns to the caller. This operation gets two shape-graphs as arguments, one from the call-site and the other from the callee exit-site.

Section F.3 realizes the aforementioned "black box" operations to realize shape analyses algorithms using the abstract semantics described in Sections 3.7.3, 4.8.3.

The tabulation algorithm propagates *path-edges*. A *path-edge* $\langle sg_{entry}, sg\rangle$ is propagated to a control flow graph node $n \in N_p$ iff there exists an interprocedural-valid-path [SP81] from $entryp$ to n such that applying the composed effect of all abstract transformers associated with statements along the path to sg_{entry} results in

sg [RHS95].

The algorithm maintains the following data structures:

- The set $PathSet(n)$ contains all path edges propagated to node n. These sets are initialized to $\emptyset$. Note that $PathSet(exitp)$ contains the (already-computed) summarized effect of the procedure. Thus, we define $sm : Entry^\star \to SG \to 2^{SG}$ which maintains the procedure summary as $sm(entryp)sg_{entry} = \{sg_{exit} : \langle sg_{entry}, sg_{exit}\rangle \in PathSet(exitp)\}$.
- The multi-map $CTXs : Entry^\star \to (SG \times SG) \to 2^{Call^\star \times SG \times SG}$ associates every procedure p, identified by its entry-site, $entryp$, with its calling context. The calling-context is a map from every 0-length path-edge $\langle sg_{entry}, sg_{entry}\rangle \in PathSet(entryp)$ which was propagated to p's entry to a set of pairs of a call-site $n_{call}^{caller} \in Call^\star$ and a path-edge $\langle sg_{entry}^{caller}, sg_{call}^{caller}\rangle \in PathSet(n_{call}^{caller})$ such that the analysis of the invocation of p at call-site n_{call}^{caller} extracted the shape-graph sg_{entry} out of sg_{call}^{caller}. This map is initialized to associate entry-nodes with empty maps.

The iterative algorithm (procedure `tabulate`) is defined in Figure F.1. The *worklist* is initialized to contain a 0-length path edge from a shape-graph representing the memory at the entry to the program to the same shape-graph. It then iterates until the *worklist* is exhausted. In every iteration, the algorithm extracts one *event* out of the *worklist*. An *event* is comprised of a CFG node n and a path edge $\langle sg_{entry}, sg\rangle$. The algorithm performs one of the following operations depending on the role of n:

- If n represents a procedure entry, a return-site or a program location of an intraprocedural statement, the algorithm *applies* the abstract transformer associated with each edge emanating from n, propagating an (extended) path edge, if necessary.
- If n represents a call-site to procedure p, the algorithm *extract*s sg', the shape-graph representing the callee's local-heap from the target of the path-edge (sg). It then adds the call-site n and path-edge $\langle sg_{entry}, sg\rangle$ to the calling contexts of $CTXs(n_{callee}^{entry})\ sg'$. This operation "registers" $\langle n, \langle sg_{entry}, sg\rangle\rangle$ as a calling-context of p, which means that whenever a *new* path edge whose source is sg' is propagated to $exitp$, the algorithm propagates an appropriate shape-graph to the return-site $return(n)$. If the path edge $\langle sg', sg'\rangle$ has not been propagated to $entryp$, the algorithm propagates it. Otherwise, the algorithm propagates the known summary effect of p on sg' to the return-site using `addToRet` (see next case).
- If n represents the exit-site of procedure p, the algorithm updates the return-site of every calling-context which is registered for sg_{entry} (i.e., in $CTXs(entryp)\ sg_{entry}$) using `addToRet`. The function `addToRet` *combine*s the shape-graph at the exit-site of the callee with the shape which is the target of the path-edge at the call-site.

The algorithm also uses the operation `propagate`$(n, \langle sg_{entry}, sg\rangle)$ that adds the edge $\langle sg_{entry}, sg\rangle$ to $PathSet(n)$, the set of path-edges at n; and inserts the event $\langle n : \langle sg_{entry}, sg\rangle\rangle$ to the worklist.

F.3 Tabulation-Based Interprocedural Functional Shape Analysis

We utilize the tabulation algorithm defined in Figure F.1 for the analyses described in Sections 3.8, 4.9 using different instantiations for the operations *apply*, *applicable*, *extract*, and *combine*.

F.3.1 Tabulation Algorithm for Section 3.8

We obtain a tabulation algorithm for the analysis framework described in Section 3.8 by instantiating the four aforementioned operations in the following way: We implement *apply* by evaluating the abstract transformer associate with the given atomic statement (see Section C.2.1.1 and Section 3.7.3). The operations *applicable*, *extract*, and *combine* are implemented by evaluating the different steps in the procedure call rule instantiated for the given call: The operation *applicable* is implemented by evaluating the side condition on the abstract memory state that arises at the call site. Thus, it (conservatively) verifies that the invocation is cutpoint-free. The operations *extract* and *combine* are implemented by following the rule's specification on how to construct the (abstract) memory states at the callee's entry state and at the caller's return site, respectively (see Section 3.7.2).

F.3.2 Tabulation Algorithm for Section 4.9

We obtain a tabulation algorithm for the analysis framework described in Chapter 4 by instantiating the four aforementioned operations in the following way: We implement *apply* by evaluating the abstract transformer associated with the given atomic statement. The operation *applicable* is treated as a function that always return true, i.e., every procedure call statement can be applied to any given shape-graph. The operations *extract* and *combine* are implemented by following the rule's specification on how to construct the (abstract) memory states at the callee's entry state and at the caller's return site, respectively (see Section 4.8.2).

"*The poets have been mysteriously silent on the subject of cheese.*"

–G. K. Chesterton