End-User Program Analysis

by

Bor-Yuh Evan Chang

B.S. (Carnegie Mellon University) 2002
M.S. (University of California, Berkeley) 2005

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor George C. Necula, Chair
Professor Koushik Sen
Professor Jack Silver

Fall 2008

The dissertation of Bor-Yuh Evan Chang is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2008

End-User Program Analysis

Abstract

End-User Program Analysis

by

Bor-Yuh Evan Chang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor George C. Necula, Chair

Program analysis tools are being adopted by industry to improve the reliability and overall quality of software like never before because they can rule out entire classes of errors. Yet, today's tools are far from being as effective or as broadly applied as they could be because for the most part, they are considered expert tools.

Because of fundamental limitations in what can be computed automatically, all program analyzers must incorporate some amount of domain-specific knowledge. Typically, such domain-specific knowledge is either provided by expert-user specification or built directly into the analyzer. The former approach often expects a high-level of program analysis expertise, while the latter does not allow for any input from the program developer—the person who best understands the code being analyzed. Instead, we advocate a more flexible view where we look to users to cooperate with the analyzer but without expecting them to be program analysis experts. That is, we make progress towards *end-user program analysis*

where non-experts can interact with an analyzer to provide the domain-specific knowledge it needs in order to give users the analysis results they want.

In particular, this dissertation presents a new technique, based on the end-user approach, for precise program analysis in the presence of data structures. Program analyzers that reason precisely about data structures have typically required sophisticated (and thus often burdensome) logical invariant specifications from the user. Instead, we propose a novel way to involve the user in guiding the analysis by extracting both the necessary invariants and reasoning rules from executable assertions.

Our technique is based on data structure validation code that is often written anyway for testing purposes. From the developer's perspective, such validation code provides guidance to the analysis in a familiar style, and we show how our analysis results can be rendered graphically in a form that is comparable to what might be drawn on a whiteboard or printed in a textbook. From the analysis tool's perspective, data structure validation code provides the essential ingredients for a good abstraction that precisely represents the important facts while ignoring irrelevant details. The crucial innovations in our system are automatic methods for understanding and generalizing the developer-provided data structure validation code specifications in order to make them useful for static program analysis.

Professor George C. Necula
Dissertation Committee Chair

To Dad and Mom

for all of the sacrifices they have made on my behalf

To Ana

for her ceaseless love and support

Contents

List of Figures

List of Tables

List of Examples

Acknowledgments

I am extremely grateful to the many people who have helped me complete this dissertation, either directly or indirectly. With absolute certainty, I am deeply indebted to my friend and advisor, George Necula. His keen technical insights, thoughtful career advice, unwavering confidence in my abilities, and continuous support of my development have been instrumental during my six years in graduate school. I cannot possibly list all of the ways that George has helped me to mature into a better researcher. I only hope that some of his many talents has rubbed off on me in the past six years, and I am certain I will be looking to his example for many years to come.

The work in this dissertation has been improved greatly through collaboration and discussion. Xavier Rival has been a tremendous collaborator who has made invaluable contributions to both the theory and implementation of this work. I cannot count the number of times we have worked through problems together (and it was always particularly helpful to get a French perspective). I benefited a great deal from Bill McCloskey and Gilad Arnold who provided insightful comments, critiqued drafts, and offered their unique perspectives on the shape analysis problem. I have also had the opportunity to have fruitful discussions about this material with many others, including Mooly Sagiv, Hongseok Yang, and Byron Cook.

I also appreciate the efforts of Koushik Sen and Jack Silver who served on my dissertation committee. Additionally, Ras Bodík and Eric Brewer provided thoughtful feedback while serving on my proposal committee. Thanks is also owed to Sanjit Seshia who provided useful advice many times and particularly about the job search.

Being part of the Open Source Quality Group has been a huge asset. I am indebted to all my friends and colleagues in the OSQ crowd who certainly shaped this work and who were always the toughest audience. Among them, I benefited greatly from earlier collaborations with Adam Chlipala, Robert Schneck, and Kun Gao on the Open Verifier. Adam provided me assistance and feedback on numerous projects and papers. I enjoyed many meals and conversations with my officemates from 565 Soda Hall (and honorary 565 occupants), including Wes Weimer, Jeremy Condit, Tachio Terauchi, Sumit Gulwani, Scott McPeak, Matt Harren, Simon Goldsmith, and Ranjit Jhala. They have been a constant source of advice and humor. A special thanks goes out to Wes not only for hosting his infamous "Wes Dinners" but also for providing guidance in navigating the job search, as well as academia in general. Let me also single out the PL crew who started at Berkeley, as I did, in 2002, including Bill, AJ Shankar, Manu Sridharan, and Dave Mandelin. They have always asked just the right questions at all of my practice talks in the past six years.

I am also grateful to those who provided me the opportunity to spend two summers at Microsoft Research in Redmond. Rustan Leino introduced me to the many problems in heap analysis and verification. At MSR, I was also fortunate enough to have had numerous stimulating discussions with many others, including Manuel Fähndrich, Mike Barnett, Wolfram Schulte, Sriram Rajamani, and Tom Ball.

I am particularly indebted to my undergraduate advisors Frank Pfenning and Bob Harper whose courses in programming language semantics and logic set me on the path I pursue today. I was extremely fortunate to have been included in the "types" group at CMU, led by Frank, Bob, Karl Crary, and Peter Lee. They introduced me to the craft of research and showed me the importance of rigor in one's work.

Last but not least, I could not have completed this dissertation without the love and support from my family. My father and mother Gwong-Jen and Li-Hua Chang have made so many sacrifices to provide me with the opportunities to get to this point, particularly the difficult choice to have immigrated from their homeland of Taiwan. Furthermore, I could always count on my father to give me the advice I needed in the toughest times. Finally, I am forever grateful to my best friend, my other half, and my wife Ana Ramírez Chang. I cannot possibly thank her enough for the strength and determination she has given me—seeing me through all the ups and downs to finish this dissertation.

Chapter 1

Introduction

Our society continues to become increasingly dependent on software, yet software systems are full of bugs. As we progress technologically, we desire better machines to ease the demands on ourselves, but as this happens, the cost of errors becomes ever greater (financially or even physically). The oft-cited 2002 National Institute of Standards and Technology report estimates the cost of software errors to the U.S. economy to be $59.5 billion annually, which is about 0.6 percent of the gross domestic product [National Institute of Standards and Technology 2002]. That amount is more than the total annual revenue of Microsoft and more than ten times the annual budget of the National Science Foundation! The case of the Therac-25, a controlled radiation therapy machine that massively overdosed six people, is a particularly frightening example of minor software errors causing catastrophic results [Leveson and Turner 1993]. As a result, the software industry is adopting program analysis tools and techniques like never before (in addition to continued emphasis in testing). For example, Microsoft distributes its Static Driver Verifier, a compile-time tool for checking that Windows device drivers adhere to certain usage rules [Ball et al. 2004]; Airbus applies the Astrée static analyzer to its safety-critical flight control software [Delmas and Souyris 2007]; and companies, like Coverity, Fortify, and GrammaTech, are marketing static source code analysis tools.

While testing remains an important technique for improving the reliability and overall quality of software, program analysis tools are being increasingly adopted because they can rule out entire classes of errors. Yet, today's tools are far from being as effective or as broadly integrated in software engineering processes as they could be because for the most part, they are still considered expert tools. Except in special circumstances, such tools are hampered by high *false-alarm* rates or *unsoundness*, that is, unless under expert control, program analysis tools often produce many spurious warnings or otherwise, choose to potentially miss actual errors.

Program analysis is the systematic examination of program code to automatically derive properties of the program or verify the absence of errors. At the core of algorithms for program analysis is a simulation of the program to derive conceptually *all* possible states the program may enter. Since this set of states is not computable in general, a common theme in program analysis design is the necessity for approximation. In other words, a program analyzer by its choice of approximation is targeted to a particular class of programs; analyzing programs that fall outside this class yields poor results, such as spurious warnings, missed errors, or outright rejection of the program.

As a result, an important criterion by which program analyzers are evaluated is their *precision*. Precision relates to how large or varied is the class of programs that can be effectively analyzed, that is, if a particular property holds for a specific program, can the analyzer determine that it is the case? While we continuingly want better precision, we must typically weigh added precision against added complexity. This added complexity may be in terms of increased computational cost (e.g., to track more detailed information) or perhaps in terms of decreased usability (e.g., by requiring careful guidance from the user).

I argue that the tension between precision and complexity of analysis has traditionally led to two conflicting forces in program analysis design. On one hand, we want to be able to analyze all (or at least more and more) programs, which pushes us toward powerful and *generic* verification frameworks that often require expert knowledge to use effectively and may be difficult to scale to large programs. For example, in the limit, we provide a suitable logic (perhaps with a mechanized proof checker) and ask the user to manually write proofs of program correctness. The following statement summarizes this mentality:

> *As there are fundamental limitations in what can be decided automatically, why can't software developers write more* ***specifications for our analyzer****? Then, we could verify so much more.*

On the other hand, we want our tools and techniques to be adopted into mainstream software engineering processes, which means they must be easy to use and scale to large, "real-world" programs. This desire drives us toward "fully-automatic" analysis tools that try to shield the user from the intricate details of the analysis algorithm or logical formalism. With this mindset, we, as program analysis builders, study and try to understand a class of programs, and then, we design an analysis approximation tailored to that class. We hope this class is "interesting" or "representative" and live with poor results for programs outside this class.

The following statement sums up this reaction:

> *As software developers won't or are unable to write sophisticated specifications for our analyzer, we should design "fully-automatic" analyzers (though perhaps coarse) that hopefully work most of the time.*

While the "fully-automatic" or *specialized* approach has yielded recent successes, such as the aforementioned tools being used in industry, what should we do when we again run into the fundamental limitations? For example, it may be quite difficult to patch such an analyzer to enlarge the class of analyzable programs (i.e., to be more precise) without giving up the original usability properties of the analyzer.

1.1 Thesis Statement

My thesis is that with a slight shift in mentality toward the user of the analyzer, we can make progress toward more precise program analysis without sacrificing usability. Instead of strictly demanding either specification based on the needs and formalism of the analysis tool or nothing at all, we advocate a more flexible view. In particular, we shift our mindset to the following:

> *Analyses improve with more information about the program. Can we design program analyzers* ***around the user*** *to obtain additional information?*

As an example, software developers already write testing code to improve software quality. Can we adapt an analyzer to use those as specifications? Can we involve the user in the program analysis without expecting the user to be a program analysis expert, thereby enabling *end-user program analysis*? To be unambiguous when referring to the various human participants, I utilize the term *user* to mean the user of program analysis (as opposed to of software). Typically, the target user of program analysis corresponds to the software *developer*, as they can use analysis tools to help them better understand and eliminate errors from their code.

The focus of this dissertation is a new technique, based on the end-user approach, for precise program analysis in the presence of heap-allocated data structures (known as shape analysis). Program analyzers that reason precisely about data structures have typically required sophisticated (and thus often burdensome) logical invariant specifications from the user. Instead, we propose a novel way to involve the user in guiding the analyzer by extracting both the necessary invariants and reasoning rules from executable assertions in the code. This work targets software developers, as it examines how to take information about the data structure known to the developer and often expressed as testing code in order to reason effectively about data structure use (discussed further in Section 2.3).

1.2 A Brief Survey of Program Analysis and Verification

The general area of program analysis and verification is among the oldest in computer science. Early on, prominent figures, such as Floyd, Hoare, and Dijkstra, recognized the need for techniques and methodologies to deal with the growing complexity of computer programs (even in the 1960s and 1970s!). Over the years, this universal problem has been tackled by many, and several important techniques have been developed that have each led to distinct subareas with their own communities. As such, the aim of this section is not to provide a complete history of program analysis and verification but rather to speak broadly and touch upon the generic-specialized tension that motivates this dissertation.

Deductive Verification

At the foundation of program analysis and verification are ways to describe program behavior mathematically. Floyd [1967] and Hoare [1969] set the stage by pioneering the use of logical formulas to describe program behavior. By viewing program states as logical formulas and interpreting programs statements as predicate transformers, they provided a generic, mathematical framework for reasoning about programs (in what we call today *program logics* or *Hoare logics*). Dijkstra [1976] took this a step further by promoting the development of programs along with proofs of their correctness simultaneously. The draw of this technique is its generality; however, a major difficulty that arises is the need to specify *loop invariants*; a loop invariant characterizes the program state at a point inside the loop regardless of how many times the loop executes. Loop invariants tend to be complex and thus are notorious for being a difficult specification to provide.

Probably some of the most successful tools based on *deductive verification* perform what is called *extended static checking* (e.g., ESC/Java [Flanagan et al. 2002] and Spec$^{\#}$ [Barnett et al. 2004]) where the focus is not full verification but to check run-time assertions (e.g., null-deference, array out-of-bounds, and developer-written assertions). In such systems, user effort is mitigated by using automated theorem proving technology (though loop invariants are still required for sound verification). On one hand, these tools are generic in that they permit expressive specifications, but at the same, they are specialized in that the limitations of automated theorem proving technology confine what can be checked effectively.

Model Checking

Model checking [Clarke and Emerson 1981; Queille and Sifakis 1982] advocates a fairly different approach where the focus was originally on verifying finite-state systems by exhaustive exploration of the space of computation states according to a specification in temporal logic. Within the past ten years, there has been a surge in tools applying model checking to software through the use of abstraction (i.e., a way to summarize potentially infinite sets of computation states) and specific techniques, such as, predicate abstraction and counterexample-guided abstraction refinement (e.g., SLAM [Ball and Rajamani 2001] and BLAST [Henzinger et al. 2002]). A particularly appealing property of this method is that model checkers are generally fully-automatic and thus require little or no programmer involvement. Roughly speaking, counterexample-guided abstraction refinement is a way to "auto-tune" the analysis abstraction based on the property of interest. The aforementioned success of Microsoft's Static Driver Verifier is a direct evolution of the SLAM project. These tools are specialized to a large degree, as predicate abstraction fixes a family of abstractions over which the refinement algorithm explores. With end-user program analysis, we take a slightly different perspective where we want to involve the user in coming up with the appropriate abstraction for the program of interest.

Abstract Interpretation

If we allow the analysis builder to design a custom abstraction or representation of the program state, then we leave it to the analysis designer to ensure that the analysis captures the facts of interest and that loop invariants can be inferred. This premise is the basis of *data-flow analysis* [Kildall 1973] and *abstract interpretation* [Cousot and Cousot 1977]. Abstract interpretation provides a framework for the design of sound program analyses based on using a custom abstraction of program states. Its theory frees the analysis designer to use even unbounded abstractions provided a *widening* operator for ensuring convergence is defined. The previously mentioned Astrée static analyzer is a success that can be attributed to this approach. Through a close collaboration with industry experts from Airbus, the designers have built a tool capable of analyzing real flight-control software with few or no false alarms [Blanchet et al. 2003]. At the same time, like many abstract interpretation-based program analyzers, Astrée is very specialized to the problem domain (specifically, for control-command embedded code). When it comes to safety-critical code, it may be appropriate and feasible to have a team of analysis designers and domain experts work closely to build a custom analyzer, but perhaps not in less demanding situations. The Three-Valued Logic Analysis (TVLA) framework [Sagiv et al. 2002] is an exception to specialized abstract interpreters; it is a very generic and powerful analysis framework based on abstract interpretation. With TVLA, the user can specify the properties of interest by defining *instrumentation predicates* (in first-order logic with transitive closure), and the framework provides a mechanism to ensure convergence (*canonical abstraction*). While its allure is certainly its generality, defining the right instrumentation predicates are critically important for precision and efficiency. The primary application of TVLA has been to shape analysis, so I discuss it further in Section 2.2. With end-user program analysis, we largely take the abstract interpretation view, but we find ways to involve the user in customizing the abstraction (without expecting the user to be a program analysis expert).

Type Systems

Lastly, there has been a formalism and analysis technique based on user-supplied specifications that has seen fairly widespread adoption. Static type systems have long been part of fairly commonly used languages, like ML, but have also made their way into mainstream languages used in industry, such as Java and C#. Type systems prevent errors by classifying values into types (e.g., integers, functions) and prescribe rules that prevent operations from being performed on inappropriate types (e.g., adding an integer to a function). At a high-level, *type inference* [Milner 1978] is a program analysis where user-defined types contribute some basic amount of user involvement in abstraction. While most uses of type systems try to strike a balance between the expressiveness of the types and the ability to check them without "too much" user annotation, at its limit (e.g., in the interactive theorem proving environment Coq [Bertot and Castéran 2004]), it is a logical specification language where checking requires annotations at least akin in complexity to loop invariants. With end-user program analysis, we look to typical type systems as an example of user-friendly program-specific abstraction, but we take a more liberal view on how to involve the user to try to get, for instance, invariant inference.

1.3 Dissertation Outline

Chapter 2 provides an overview of shape analysis and our contributions (with little analysis expertise expected). In that chapter, I discuss further why reasoning about the heap and heap-allocated data structures is one of the main limitations in today's development tools and why shape analysis is a good case study for end-user program analysis. I also present an overview of our tool and techniques by following example analyses from the user's perspective. Through this process, I identify the main components of our analyzer and motivate their design. The details of the design, implementation, and evaluation of our shape analysis tool, Xisa, are then described in Chapter 3, Chapter 4, and Chapter 5. Finally, in Chapter 6, I provide some concluding thoughts, summarize the contributions of this dissertation, and propose some directions for future work.

Chapter 2

Overview: Analyzing Code with Unbounded Data Structures

Heap manipulation (e.g., via pointer updates) is fundamental in almost all software developed today, especially those developed using main-stream, large-scale, imperative programming languages, such as C, C++, Java, and C#. Tools that aim to verify properties of interest, perform complex program transformations, such as code refactoring, or provide useful information during development time often require detailed aliasing and memory structure information. Yet, one of the main weaknesses of program analyzers today is their inability to reason precisely about heap manipulation, particularly when objects of interest are put into data structures. These issues are only exacerbated by a trend towards more software being built with ever more dynamic languages, such as Perl, JavaScript, and Python.

Shape analyses examine how to define and manipulate precise heap abstractions even in the presence of unbounded data structures (e.g., lists and trees). This property makes them unique in the kind of detailed aliasing and memory structure information they can provide and thus useful in improving program analyzers today that, for example, try to eliminate memory safety errors (e.g., leaks, dangling pointers), resource usage bugs (e.g., with locks, file handles, and database connections), or concurrency errors (e.g., data races). They can even be used to perform complex program transformations, like compile-time garbage collection (e.g., Arnold et al. [2006]). Unfortunately, because of precision requirements, shape analyses have been generally prohibitively expensive to be broadly used in practice.

In the remainder of this chapter, I give some background showing some of the challenges with reasoning about unbounded data structures and highlighting some of the characteristics of shape analysis (Section 2.1). I then survey prior work on shape analysis (Section 2.2) as context for our end-user approach (Section 2.3) before providing an introduction to our tool and techniques from the user's perspective (Section 2.4). Finally, I conclude the chapter with a summary of the contributions of this work (Section 2.5).

2.1 Background: Shape Analysis

This section provides some basic background in shape analysis. The aim is to give an introduction to the problem addressed by shape analysis and capture some of its distinguishing features to help make the presentation of our analysis in subsequent sections clearer; it is not necessarily intended to be a complete treatment of the discussed concepts and terminology. Further discussion can be found in the overview texts by Reps et al. [2007]

and Nielson et al. [1999].

2.1.1 Defining Shape Analysis

As a concrete example for how shape analysis can be used, we consider a program analysis that checks for the proper usage of a locking interface. In particular, we look at checking that the program never tries to re-`acquire` a lock it has already acquired (within the same thread), that is, for any call to `acquire`, we want to ensure the following precondition:

```
assert(lock has not been acquired by this thread);
acquire(lock);
```

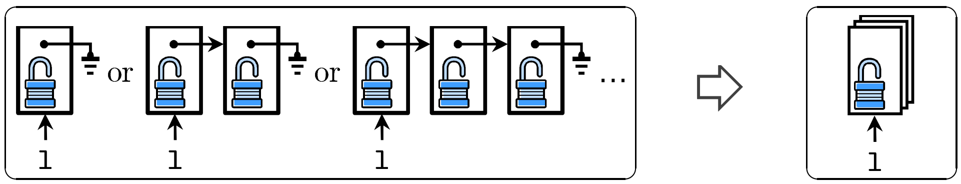
Many tools and techniques have been developed to check this kind of property (e.g., see Ball and Rajamani [2001], Henzinger et al. [2002], Das et al. [2002], Fähndrich and DeLine [2002], or Fink et al. [2008]). Microsoft's Static Driver Verifier, one of the previously mentioned industrial tools, tries to eliminate bugs of this form.

The analysis problem becomes more difficult when locks (i.e., the objects of interest) are put into data structures, and it is in this situation that such higher-level analyses could benefit from shape analysis. To illustrate the added complexity from data structures, we consider a simple function that takes a singly-linked list of unacquired locks and calls `acquire` on each lock in the list (see Figure 2.1). In Figure 2.1(b), I present the function annotated with informal graphical representations of the memory state at a few program points (shown boxed). On the left, I show an "ideal analysis state" that alludes to an unbounded number of configurations (depending on the length of the list and the loop iteration). Assuming the function takes a list of unacquired locks, the `acquire` statement

```
typedef struct LockNode { Lock lock; struct LockNode *next; } LockNode;
```

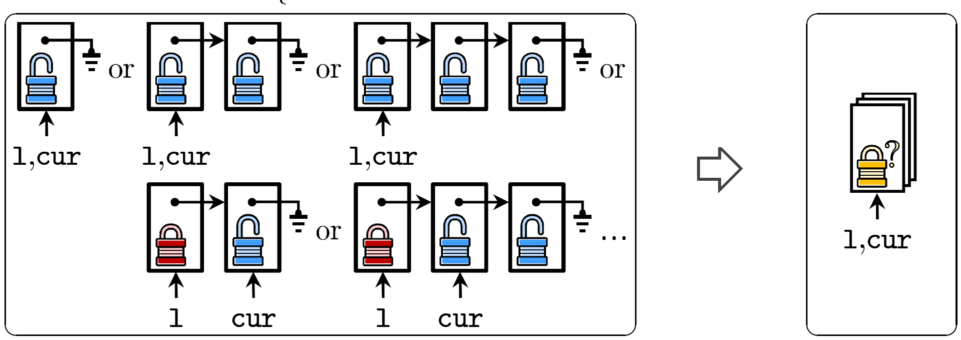
(a) A C type definition for a singly-linked list node containing a lock.

```
void acquire_all(LockNode *l) {
  LockNode *cur;
```

1 


```
  cur = l;
```

2 `while (cur != null) {`

3 


```
    assert(cur->lock has not been acquired by this thread); acquire(cur->lock);
```

4 `cur = cur->next;`

5 `}`

6 `}`

ideal analysis state *abstraction without shape*

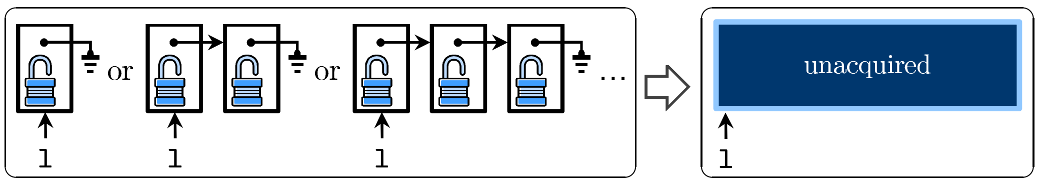
(b) A function that acquires a list of locks in C.

Figure 2.1: An example to illustrate some challenges in analyzing unbounded data structures. The code in (b) is annotated with informal graphical representations of the memory state at a few program points (shown boxed). The memory states on the left depict "ideal analysis states" where no information is lost with respect to the data structure. In those pictures, the boxes represent linked-list nodes and are labeled with whether the state of the lock is unacquired (light/blue) or acquired (dark/red). The memory states on the right show a typical abstraction for data structures that collapse all nodes into one abstract object. An **assert** statement is included at program point 3 to emphasize the "no double `acquire`" property that we are trying to check.

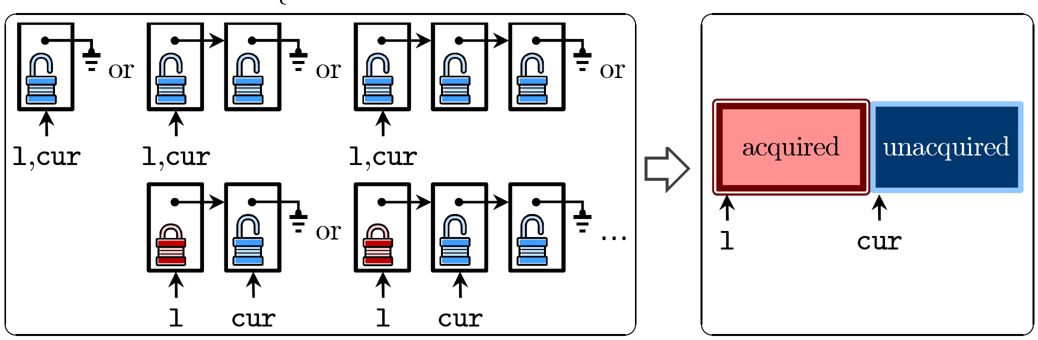
at program point 3 does indeed respect the locking protocol (as can be seen in the "ideal analysis states"). Unfortunately, since we cannot, in general, represent an unbounded set of configurations, a program analysis must perform some kind of abstraction (notated with the large arrow in Figure 2.1). A typical abstraction for data structures is where all nodes of the data structure are represented by one *abstract object* (as shown on the right). At program point 3, because the locks in the list may be unacquired or acquired, this abstraction can only say that the abstract object (representing all elements of the list) may be unacquired or acquired. As we cannot distinguish the unacquired nodes from the acquired ones, the analysis has lost the information that `cur->lock` is in fact an unacquired lock. To be sound, such an analyzer must conservatively report that the `acquire` statement as potentially buggy—raising a false-alarm. The issue is a matter of precision; this abstraction is not precise enough to capture that `cur->lock` is always an unacquired lock. No program analysis can be perfectly precise, but this example shows a situation that could benefit from shape analysis. Here, we have an abstraction that does not take into account any shape information; for example, the abstraction would be same if we instead had a tree of locks.

In Figure 2.2, we consider informally an abstraction that takes into account shape information for analyzing the `acquire_all` code. At program point 1, we assume that `l` points to a memory region of linked-list nodes with unacquired locks, capturing basically the same information as the abstraction in Figure 2.1. However, at program point 3, this analysis with shape information splits that region into two parts: one between `l` and `cur` where the locks have been acquired and one from `cur` where the locks are still unacquired. In particular, we have that `cur->lock` is unacquired at that point giving us enough precision

```
void acquire_all(LockNode *l) {
  LockNode *cur;
```




```
  cur = l;
  while (cur != null) {
```




```
    assert(cur->lock has not been acquired by this thread); acquire(cur->lock);
    cur = cur->next;
  }
}
```

ideal analysis state *abstraction with shape*

Figure 2.2: A revision of the example in Figure 2.1 that uses an analysis abstraction with shape information. The wide rectangles represent singly-linked list regions (i.e., segments) of locks, which may be unacquired (dark fill/blue) or acquired (light fill/red).

to determine that the `acquire` statement is indeed fine.

Typically speaking, shape analysis is concerned with providing precise information about the arrangement of heap-allocated data structures (e.g., a variable points to an acyclic singly-linked list, a cyclic doubly-linked list, or a binary tree). That is, the focus is on pointer values rather than scalar values, and pointer properties (e.g., nullness, may-aliasing, must-aliasing, sharing, and cyclicity) are constituent properties of or derived facts from shape invariants. With this example, we have taken a more inclusive perspective mixing *shape properties* (e.g., `l` points to a singly-linked list) and *data properties* (e.g., `cur->lock` is unacquired). We see that the shape information serves as a foundation for checking the higher-level locking property.

From this example, we observe some of the characteristics of shape analysis. Shape analysis, by its nature, is *flow-sensitive*, that is, it respects the control-flow of the program and the execution order of statements. In the examples, the flow-sensitivity is indicated by showing distinct abstract memory states at different program points. One distinguishing feature of present-day shape analysis is that the *heap abstraction*, that is, the partitioning of memory into regions, also depends on the program point (contrast Figure 2.1 and Figure 2.2). In Figure 2.1, the abstraction of the list nodes into one region—one abstract object—is the same at all program points even though the analysis is flow-sensitive. Figure 2.1 shows, in essence, an `acquire` checker built on a typical pointer analysis where the heap abstraction is fixed. There has been work that tries to recover some amount of precision from such fixed heap abstractions by integrating with the high-level property of interest (e.g., Fink et al. [2008]). Instead, we will look directly at the shape analysis problem and explore how to make it more practical by taking the end-user approach.

To summarize, a *shape analysis* computes for a given program, at each program point,

> *a bounded abstraction of the memory state with distinguishable descriptions of the heap-allocated data structures.*

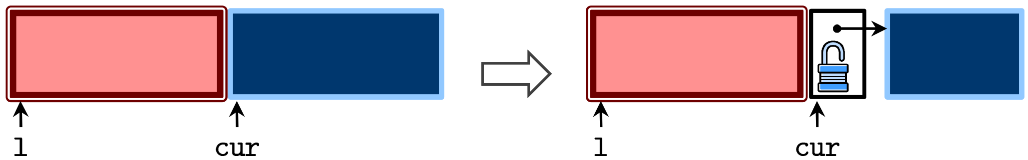
Shape analyses vary widely in their heap abstraction and how they derive invariants (see Section 2.2). Yet, there are some common underpinnings in how they obtain precise heap descriptions on a per program point basis. In the next subsection, I sketch the key operations common to many shape analysis algorithms (including ours) to set the stage for describing how they are instantiated in our design.

2.1.2 Shape Analysis Operations

We consider shape analysis in the context of a standard forward *abstract interpretation* [Cousot and Cousot 1977]. A local invariant (e.g., an abstract memory state) is computed automatically for each program point by interpreting program statements according to an *abstract semantics* (i.e., by simulating the program on abstract memory descriptions). Provided an appropriate abstraction is set up, we get a high degree of automation, as there is no need to manually describe complicated intermediate states (e.g., loop invariants). Alluding to our approach (Section 2.3), the user is, in essence, involved in customizing the shape abstraction so that it is appropriate for the program being analyzed.

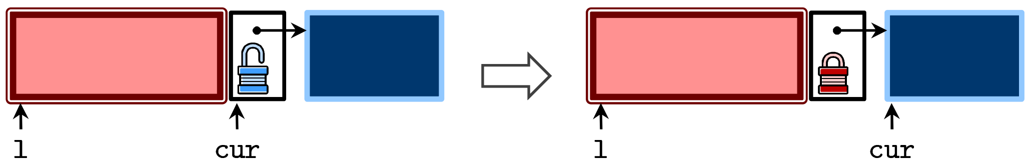
To analyze a program, we need *abstract transformers* that take an abstract memory state and alter it appropriately to reflect update statements. In shape analysis, the heap abstraction gives a finite partitioning of memory into regions where each region is a *summary*

Materialization: Splitting of summaries.



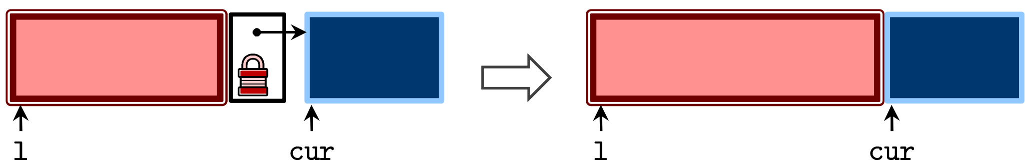
(a)

Precise Update: Reflecting state change precisely.



(b)

Summarization: Consolidating for termination.



(c)

Figure 2.3: Key operations in shape analysis. These diagrams refer loosely to the example presented in Figure 2.2.

of a number of possible concrete memory cells. A challenge in designing a shape analysis is to define a way to reflect update statements on summaries precisely. Figure 2.3 presents schematically the key components in many shape analysis algorithms (including ours) that makes this precise reasoning possible.

Materialization

To reflect state change precisely, contemporary shape analyses first perform a splitting of summaries known as *materialization*, which exposes concrete cells locally around where an update applies. In Figure 2.3(a), I show a materialization of a concrete node at the cursor `cur` from the summary region. From the analysis point of view, this refinement step is crucial for precision (e.g., in the analysis of traversals through pointer-based data structures). Chase et al. [1990] had observed that keeping instances of concrete cells separate from summaries is critical for precisely analyzing data structure operations. Sagiv et al. defined an early materialization operation to obtain concrete instances from summaries [Sagiv et al. 1998], and then made a further innovation by viewing the operation as "partial concretization" [Sagiv et al. 2002].

As shown, this step is much like what we might expect to see in whiteboard drawings during a manual code review. We intuitively understand what singly-linked lists look like and that the dark blue box is a singly-linked list of unacquired locks. Whereas, for an automated shape analysis, the challenge is how does it carry out such a splitting operation and how does the analysis know where to apply it.

Precise Update

Once concrete cells have been materialized, updating the memory state seems fairly straightforward. In Figure 2.3(b), I depict a change of the lock state and the advancement of `cur` (i.e., the result of interpreting the loop body of the example in Figure 2.2 from program point 3 to point 5).

Note that the materialization operation enables the shape analysis to perform *precise updates* in that the update of the analysis state applies to exactly one concrete cell. Enabling this property is what makes materialization crucial for precision. We observe that without materialization, the best invariant we could hope for in this case is where we have that `l` and `cur` point to a region with locks that could be either acquired or unacquired, which is basically the invariant at program point 3 in Figure 2.1.

Summarization

With materialization, in order for a shape analysis to converge, it needs to have a *summarization* operation that "re-summarizes" concrete cells—in an informal sense, an inverse to materialization (as shown in Figure 2.3(c)). For instance, with just materialization while analyzing the loop in our example program (Figure 2.2), we could always materialize at `cur` and consider the case of a linked list with one more node on each iteration (i.e., computing an abstraction for each row of the "ideal analysis state" on the left). The challenge in designing a summarization operation is how to prevent losing too much precision while ensuring termination.

2.2 State of the Art in Shape Analysis

Shape analysis has long been an active area of research with numerous algorithms proposed and systems developed. In this section, I provide an overview of the main techniques to set up a comparison with our end-user approach in the next section (Section 2.3). The focus in this section is on shape invariant inference techniques, though I mention some related work on heap reasoning in general at the end.

An early work on shape analysis for languages with destructive update was that of Jones and Muchnick [1981]. Their analysis abstracted memory as sets of *shape graphs* where a node corresponded to an access path from a program variable (i.e., a sequence of field reads from a variable). To ensure finiteness of the shape graphs, they introduced the k-*limiting* approximation where access paths with length greater than k are truncated. Cells reachable only by longer accessed paths are collected into one *summary node*. Summary nodes correspond to memory regions of basically arbitrary structure, though they did carry two pieces of additional information: whether the region may contain sharing (i.e., has a cell that may be pointed to by more than one other cell) and whether the region may contain a cycle. Stated differently, this analysis along with the sequence of work that followed it based on k-limiting were largely agnostic to the particular data structure. This trait meant they lost most information on code that traversed and manipulated cells beyond the k-limit (e.g., they could not show that a list was obtained after a destructive list reversal operation). As alluded to in the last section (Section 2.1), this kind of precision requires materialization, which means the analysis needs to be data structure-specific to some degree.

TVLA [Sagiv et al. 2002] is a well-known, very powerful and generic framework based on three-valued logic and is probably one of the most widely applied shape analyzers for verifying deep properties of complex heap manipulations (e.g., Loginov et al. [2006] and Lev-Ami et al. [2000]). The framework is parametric in that users provide specifications (*instrumentation predicates*) that affect the kinds of structures tracked by the tool. Instrumentation predicates are defined in terms of predicates describing the nodes pointed to by program variables and the connectivity of the nodes given by the fields (the core predicates) using first-order logic with transitive closure. For example, common and important instrumentation predicates include reachability and sharing properties:

$$\mathsf{reach}_{\mathsf{x},\mathsf{next}}(n) \stackrel{\text{def}}{=} \mathsf{x}(n) \;\vee\; \exists n_0.\, \mathsf{x}(n_0) \wedge \mathsf{next}^+(n_0, n)$$

$$\mathsf{shared}_{\mathsf{next}}(n) \stackrel{\text{def}}{=} \exists n_1, n_2.\, \mathsf{next}(n_1, n) \wedge \mathsf{next}(n_2, n) \wedge n_1 \neq n_2$$

where x indicates a cell that is pointed to by a program variable `x` and next is a binary relation that says the `next` field of the first cell points to the second cell[1]. Instrumentation predicates allow the analysis to keep some data structure-specific properties about summaries so that the information can be used when materializing. While flexible and expressive, a large amount of follow-on and ongoing work on this topic addresses improving scalability. For instance, as instrumentation predicates may reference anywhere in the global memory state, eliminating infeasible memory states (the coerce operation) can be expensive, though recent techniques have made significant improvements [Bogudlov et al. 2007]. Yahav and Ramalingam [2004] partition the memory state into regions that are either tracked more precisely or less precisely depending on their relevance to the property in question. Manevich et al. [2004] describe a strategy to merge memory states whose canonicalizations are "similar" (i.e., have isomorphic sets of individuals). Arnold [2006] identifies an instance where a more aggressive summarization loses little precision (by allowing summary nodes to represent zero-or-more concrete nodes instead of one-or-more). Manevich et al. [2002] examine how to compactly represent this style of logical structures. Additionally, motivated at least in part by the efficiency cost of TVLA's generality, there has been work on specialized analyzers based on shape graphs that integrate properties for particular kinds of data structures (e.g., Lev-Ami et al. [2006]). Separately, there also has been some work on specialized shape analyzers (e.g., for singly-linked lists and trees) based

[1] While I have provided the defining formulas to provide some intuition, understanding their exact meaning is not necessary at this point.

on encoding into predicate abstraction frameworks (e.g., Balaban et al. [2007])

In the past few years, there has been a growing interest in shape analyses based on inductive definitions in separation logic. Most of the work in this area builds into the abstraction particular inductive definitions that correspond to data structures of interest (along with materialization and summarization rules). The first version of Space Invader [Distefano et al. 2006] summarizes singly-linked list regions using an inductive predicate for non-empty list segments:

$$\mathsf{ls}(b, e) \quad \stackrel{\text{def}}{=} \quad (b \mapsto e \ \vee \ \exists n.\, b \mapsto n * \mathsf{ls}(n, e)) \ \wedge \ b \neq e$$

where b and e are the beginning and ending cells of the list segment, respectively[2]. Berdine et al. [2007] have extended this framework to apply to doubly-linked lists polymorphically. Specifically, they infer invariants for nested list data structures. While the outer list structure is built-in (as a higher-order inductive predicate), they infer automatically a description for the shape of the "list node". Magill et al. [2007] have also extended this approach to lists with a length parameter, which allows them to check programs where safety depends on the length of the list. Calcagno et al. [2006] demonstrate how to analyze a memory allocator, showing that inductive definitions in separation logic can accommodate conversions between a word-level view and an object-level view.

Lee et al. [2005] propose a shape analysis where memory regions are summarized using grammar-based descriptions (which are then formalized as inductive predicates in separation logic). In contrast to the analyses described in the last paragraph, their analysis

[2]While understanding the exact meaning of this predicate is also not necessary at this point, it says that a list segment is composed of either one memory cell with address b and contents e (i.e., b points to e) or one-or-more cells where the first has b points to n and separately the remaining ones form a list segment from n to e (with $b \neq e$).

derives these descriptions automatically from the construction of the data structure (for a certain class of tree-like structures). Guo et al. [2007] describe a global shape analysis that also synthesizes inductive shape invariants from construction patterns present in the code. The class of shapes they consider is larger, and their data structure descriptions are directly inductive predicates in separation logic.

The abstract interpretation-based analysis proposed by Gulwani and Tiwari [2007] is based on an encoding of shape invariants into rather expressive $\exists\forall$ quantified formulas. The existential essentially specifies an endpoint for a memory region, while the universal is used to summarize the cells between the beginning of the structure and the endpoint. The technique does not make use of any explicit separation and thus requires may and must alias information to be recomputed on the fly; however, it does make it easier to integrate with standard numerical domains to reason about arrays.

In a rather different direction, Hackett and Rugina [2005] present a shape analysis that first partitions the heap using region inference and then tracks updates on representative heap cells independently. While their abstraction cannot track certain global properties like the aforementioned shape analyses, they make this trade-off to obtain a very scalable shape analysis that can handle singly-linked lists. Cherem and Rugina [2007] have extended this analysis to handle doubly-linked lists by including the tracking of neighbor cells at a very reasonable cost. However, it is not clear how to extend the analysis to more global properties amenable to the shape analyses mentioned previously.

Other Heap Reasoning Techniques

There is a vast literature pertaining to reasoning about the heap and data structures within each of the main areas of program analysis and verification discussed in Section 1.2. Here, I point out some of this work, focusing on aspects most related to our approach.

Pointer analysis serves a similar purpose as shape analysis by providing aliasing information to higher-level client analyses. While there is a large variance in the precision of pointer analysis algorithms, the distinguishing feature between pointer analysis and shape analysis is the model of the heap (as alluded in Section 2.1). Pointer analysis algorithms typically use a fixed partitioning of memory based on the static allocation site. Hind [2001] provides a survey of pointer analysis techniques. A kind of intermediate precision approach (i.e., a lightweight shape analysis) has been described by Ghiya and Hendren [1996] where path matrices are used to infer one level of shape information.

Reasoning about arrays is related to shape analysis over pointer-based data structures in that the array cells must be summarized. To derive relations between the contents of array cells, there must also be a partitioning into summaries and concrete cells. Some inference techniques that can derive relations about the contents of array cells include Gopan et al. [2005], Gulwani et al. [2008], and Halbwachs and Péron [2008].

Many deductive verification techniques have also been applied to pointer-based data structures. These techniques are typically based on verification-condition generation and thus require loop invariants. The Pointer Assertion Logic Engine (PALE) [Møller and Schwartzbach 2001] is such a system based on monadic second-order logic. Wies et al.

[2006] have extended PALE with non-deterministic field constraints (and some loop invariant inference), which enables some reasoning of, for example, skip list structures. Also, in this category, McPeak and Necula [2005] have identified a class of local equality axioms in first-order logic that can describe many common data structure invariants and have given a complete decision procedure for this class. Chatterjee et al. [2007] provide a logic with a reachability predicate but are able to discharge most proof obligations using a first-order theorem prover through a novel axiomatization. There are also a few verification systems based on inductive definitions in separation logic [Berdine et al. 2005; Nguyen et al. 2007].

There has also been work on language-based approaches where a specialized language is developed in conjunction with a type system that permits the specification of richer data structure invariants (than in typical type systems). Shape types [Fradet and Métayer 1997] allow for the specification of tree-like data structures with back and cross pointers in a fairly compact manner. For checking, the programs must be written in a specialized language Shape-C (which can then be translated to C). The aforementioned analysis of Lee et al. [2005] essentially looks at inferring shape type descriptions. In functional languages, there has been work in defining restricted forms of dependent type systems (e.g., refinement types) that can capture some data structure invariants (e.g., Xi and Pfenning [1999], Dunfield [2007]). One question is whether it is necessary to provide specifications of temporary invariant violations that depend on the code to be checked.

2.3 Approach: Invariant Checker-Based Shape Abstraction

As we saw in Section 2.1, shape analysis and program analysis in the presence of heap-allocated data structures in general demand a high degree of precision. For instance, we see this need for precision manifest in the program point-specific heap abstraction in Figure 2.2. Broadly speaking, this need for precision in shape analysis has only accentuated the divide between generic and specialized approaches to program analysis with different kinds of successes. As an example, under expert control, the TVLA system has been used to verify correctness of exceedingly complex algorithms (e.g., the Deutsch-Schorr-Waite stackless tree traversal algorithm [Loginov et al. 2006]). At the same time, the Space Invader tool has very recently been applied to check pointer safety of reasonably-sized, list-manipulating Windows and Linux device drivers (≈1,000–10,000 lines of C code) [Yang et al. 2008]. With such a sharp divide, we see an opportunity for end-user shape analysis. In particular, the kind of data structures used (e.g., singly-linked list, doubly-linked list, binary tree, skip list) and the kinds of properties of interest (e.g., shape, sortedness, balance) are fundamentally program-specific. Consequently, shape analyses need and stand to benefit from increased developer involvement.

Research Question. A shape analysis needs to build abstractions that are data structure-specific. For an end-user shape analysis, we would like to obtain data structure descriptions from the developer that specify the kind of shape and data properties of interest without expecting the user to be a program analysis expert. The research question is what are appropriate and useful end-user data structure specifications, and how can a static shape analysis be designed to take advantage of them.

Our Answer. Our approach is based on using run-time validation code as a specification for static analysis. The key observation we make is the following:

> *Developers often provide the building blocks for such program-specific shape abstractions in the form of data structure validation code.*

Data structure validation code, or a data structure *invariant checker*, is a traversal through memory that checks that the data structure has the expected shape and data properties. As a simple example, we have the following singly-linked list checker (expressed in an untyped, pseudo object-oriented notation):

Example 2.1 (A singly-linked list checker definition)**.**

```
l.list() := if (l = null) then true
            else l.next.list()
```

In this notation, I distinguish a *traversal parameter* (i.e., the root pointer `l`) that is the cursor used to recursively traverse the data structure. This checker simply walks along `next` fields until reaching `null` where it returns `true` and does little "checking" per se. For an example of a more involved shape, Figure 2.4 presents a checker for a (two-level) skip list [Pugh 1990]. The skip list checker can fail if, for instance, the `skip` field of a level 0 node is not `null`.

Such data structure validation code is, at times, already written and used by developers for testing or dynamic analysis. They have even been championed in undergraduate programming methodology textbooks (e.g., `repOk` methods in Liskov and Guttag [2000]). Our technique looks at taking advantage of such "specifications" in static analysis. We will

typedef

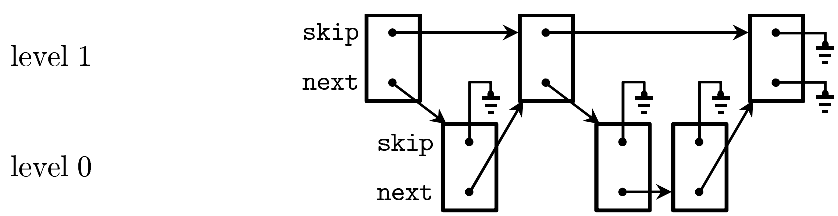
  struct `SkipNode` { `int data;` **struct** `SkipNode *skip, *next;` }

  `SkipNode;`

(a) A C type definition for a two-level skip list node.

`l.skip1()`	:=	if `(l = null)` then true
		else `l.skip.skip1()` and `l.next.skip0(l.skip)`
`l.skip0(end)`	:=	if `(l = end)` then true
		else `l.skip = null` and `l.next.skip0(end)`

(b) A two-level skip list checker definition.



(c) An example instance of a two-level skip list (satisfies `skip1`).

Figure 2.4: Describing a two-level skip list. The `skip1` and `skip0` checkers in (b) together walk over and specify the shape of two-level skip lists. In such a skip list, each node is either level 1 or level 0. All nodes are linked together with the `next` field, while the level 1 nodes are additionally linked with the `skip` field. The `skip` field of a level 0 node should be `null`. Note that to keep the definitions compact, these checkers do not specify that the last node has to be a level 1 node (as is sometimes required); this constraint could be specified with an additional case in `skip1`.

need restrictions on the form of the checkers, which then impact the kinds of data structures that can be expressed (see Section 3.2). There will necessarily be a give and take between what the developer wants to write that can easily be checked dynamically and what is feasible in our static analysis. Nonetheless, from the developer's perspective, checkers provide guidance to our shape analysis in a familiar style, and because of the guidance, we show how our analysis results can be rendered graphically in a form comparable to what might be drawn on a whiteboard or printed in a textbook (see Section 2.4).

In some respects, inductive definitions, like those given by data structure invariant checkers, are a natural fit for shape analysis, seemingly evidenced by the many shape analyses being built around them (e.g., Distefano et al. [2006], Berdine et al. [2007], Magill et al. [2007], and Guo et al. [2007]). As described in Section 2.1.2, a key component of shape analysis is materialization. With inductive definitions, a natural materialization operation is to consider unfolding the definition; in terms of validation code, we can imagine unfolding one recursive step of the checker execution. In brief, a nice property of using invariant checkers "as specification" is that they are not only a familiar way for the developer to describe the data structure invariants but also express developer intent on how the data structure should be used.

Our approach essentially strikes a balance between the generic and the specialized views. TVLA [Sagiv et al. 2002] derives its generality and expressiveness in large part because it is parameterized by fairly low-level, analyzer-oriented specifications (i.e., the instrumentation predicates) that affect the kind of structures and properties tracked by the tool. An advantage of instrumentation predicates is that they capture data structure properties component-wise. As such, they describe data structure invariants somewhat indirectly, which may make it easier to represent "intermediate states" where the data structure invariant may be temporarily broken or to capture different views of the same structure. At the same time, defining the appropriate instrumentation predicates are crucially important to the precision and efficiency of the analysis, and doing so requires a pretty good understanding of the system. So much so that there has been follow-on work looking at improving the usability of TVLA with respect to instrumentation predicates. Reps et al. [2003] provide a technique to automatically update instrumentation predicates upon applying an abstract transformer (rather than user-specified update formulas as in the original system). There has even been work to come up with the appropriate instrumentation predicates automatically (via learning techniques [Loginov et al. 2005] or backwards analysis [Ramalingam et al. 2002]). In contrast, our approach is extensible in high-level, developer-oriented specifications (i.e., the invariant checkers). These descriptions are likely to be less generic in representing memory states and thus poses a different set of challenges, but the directness of these specifications focuses the analysis to developer notions (and thus, minimizing case analysis).

On the other hand, specialized shape analyzers often build in compact, high-level data structure abstractions of interest (e.g., lists). This choice allows the analysis algorithm to be optimized around these built-in structures, which yield quite efficient analyzers. At the same time, pressure to extend such analyzers to accommodate more and more varied kinds of structures may make the abstractions less program-specific and thus potentially negate the original efficiency advantage from specialization. Instead, our approach uses compact, high-level data structure abstractions obtained from the developer. The key distinction between our analysis algorithm and those presented by, for example, Distefano et al. [2006] and Magill et al. [2007] is that inductive definitions of particular data structures are not built-in. Yet, with our approach, because the invariant checkers from the user are program-specific, we still can get good efficiency. In contrast to Lee et al. [2005] and Guo et al. [2007], our approach is to focus the shape analysis based on developer intent (rather than guess the kind of shapes of interest). Additionally, we consider checkers that include intertwined data constraints, which seem very difficult to infer generically.

Our approach presents different challenges with respect to defining the key shape analysis operations (which I discuss further and address in Chapter 3 and Chapter 4). First, we aim to build an abstraction directly out of the user-supplied invariant checkers. Yet, as alluded to earlier, while invariant checkers capture precisely the steady state of the structure as intended by the developer, we need a way to describe "intermediate states" where the data structure invariant may be temporarily broken in order to perform static analysis. Second, there is not necessarily enough information in developer-provided testing code for it to be an effective specification for static analysis. In particular, the checker definition essentially provides one materialization axiom, and this one axiom may not be enough for code using data structures with back pointers (e.g., doubly-linked lists and trees with parent pointers). Third, we need a summarization operation for arbitrary data structure checkers. For comparison, Space Invader uses hand-crafted rewriting rules tailored for the data structure kind of interest (i.e., the *abstraction rules*) [Distefano et al. 2006]. TVLA has a fixed method for ensuring boundedness (i.e., *canonical abstraction*) that work well with instrumentation predicates (which in turn makes materialization more involved) [Sagiv et al. 2002].

2.4 An Example Analysis

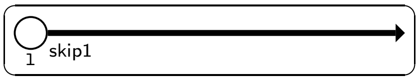
In this section, we consider our end-user shape analyzer from the user's perspective as an introduction to our technique. Figure 2.5 presents an example analysis that goes over a skip list "rebalancing" operation (i.e., an operation that reassigns each node to be either level 1 or level 0) to verify that it preserves the skip list structure. This example uses the skip list checkers defined in Figure 2.4, which the user provides. For emphasis, I show an **assert** at the top that ensures `l` is a skip list (i.e., `l.skip1()` holds) and an **assert** at the bottom that checks `l` is again a skip list on return[3]. We have made explicit these pre- and postconditions here, but we can imagine a system that connects the checker to the type and verifies that the structure invariants are preserved at function or module boundaries.

Upon running the analyzer, the result is program code annotated with abstract memory states presented in a graphical notation, which I show at a number of program points. For the program points inside the loop, there are two memory states shown: one for the first iteration (left) and one for the fixed point (right). To name heap values (e.g., a memory address), the analysis introduces *symbolic values* (i.e., fresh existential variables from the analysis perspective). To distinguish them from program variables, I use lowercase Greek letters ($\alpha, \beta, \gamma, \ldots$). A graph node denotes a heap value and, when necessary, is labeled by a symbolic value. I write a program variable (e.g., `l`) below a node to indicate

[3] The checker calls (shown underlined) indicate conceptually dynamic checks of the data structure invariant; though, it is not actually valid C code, and we do not yet have a compilation of our checker definitions into executable code.

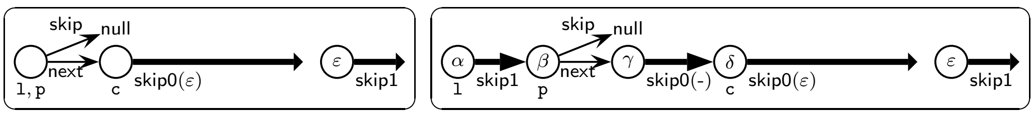
```
    void rebalance(SkipNode *l) {
      SkipNode *p;    // previous level 1 node
      SkipNode *c;    // cursor
1     assert(l != null && l.skip1());
```

2



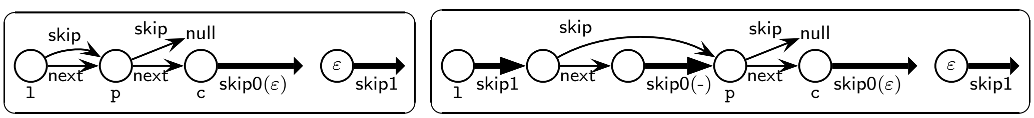

```
      p = l; l->skip = null; c = l->next;
3     while (c != null) {
```

4



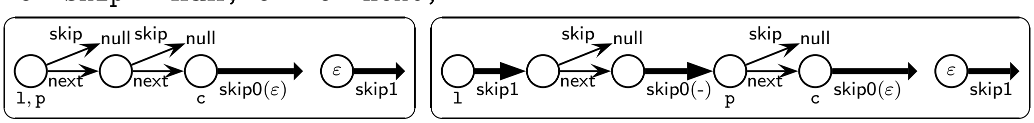

```
        if (c should be a level 1 node) {
5         p->skip = c;    // set the skip pointer of the previous level 1 node
6         p = p->skip;
7         c->skip = null; c = c->next;
```

8




```
        }
        else {
9         c->skip = null; c = c->next;
```

10




```
        }
11    }
12    assert(l != null && l.skip1());
13  }
```

first iteration — *at fixed point*

Figure 2.5: An example analysis of a skip list rebalancing. The user provides the skip list data structure invariant checker definitions shown in Figure 2.4.

that the value of that variable is that node. Edges describe disjoint memory regions[4]. A thin edge gives a *points-to* relationship, that is, a memory cell whose address is the source node with a field offset and whose value is the destination node (e.g., at program point 4 in the left graph, the edge labeled by `next` says that `l->next` points to `c`). A thick edge summarizes a memory region, that is, some number of points-to edges with certain properties. Our abstraction is built directly out of user-supplied checker definitions in that these summaries are defined in terms of them. Intuitively, a developer-defined checker can be used for summarization by viewing the memory addresses it would dereference during a successful execution as describing a class of memory regions arranged according to particular constraints. There are two kinds of *checker edges* (i.e., thick edges): complete checker edges, which have only a source node, and partial checker edges, which have both a source and a target node. Complete checker edges indicate memory regions that satisfy particular checkers (e.g., at program point 2, the complete checker edge labeled `skip1` says the memory region from `l` satisfies the skip list checker). To describe memory states at intermediate program points, partial checker edges capture the notion of a checker that holds on just a segment of a data structure (see Section 3.3 for details). For example, at program point 4 in the fixed-point graph, the partial checker edge from `l` to `p` summarizes a memory region that is a `skip1` along that segment. While these graphs are explicit enough for an analysis to use, they still seem quite close to informal sketches a developer might draw to check the code by hand. The primary difference in contrast to the pictures in Figure 2.2 is that the graphs make explicit the endpoints of the memory regions.

[4]This view differs slightly from most "shape graphs" found in the literature. In most shape graphs, nodes rather than edges represent memory cells or regions. Whereas, we use nodes to delineate the boundaries of memory regions and to distinguish between the address and the contents of memory cells.

To reflect memory updates in the graph, the analyzer simply modifies the appropriate points-to edges. For example, consider the transition from program point 4 to point 8 and the updates on line 5 and line 6. Sometimes, checker edges are *unfolded* to materialize the points-to edges for an update. For instance, observe that we do not have points-to edges for `c->skip` or `c->next` in the graph at program point 4 in order to reflect the updates on line 7. However, we have that from `c`, there is a zeroth-level skip list (i.e., an instance of `skip0` holds). It can be unfolded to materialize the `skip` and `next` fields (that is, by conceptually unfolding one step of its execution). The updates can then be reflected after unfolding.

As an end-user program analysis and as exemplified here, we want the work performed by our shape analysis to be close to the informal, on-paper verification that might be done by the developer. The abstractions used to summarize memory regions are developer-guided through the checker specifications. While it may be reasonable to build in generic summarization strategies for common structures, like lists (cf., Distefano et al. [2006] and Magill et al. [2007]), it seems less likely that other, less common structures should be built-in, like the skip lists in this example.

2.5 Algorithm Preview and Contributions

From the example in Figure 2.5, we make some observations that guide the design of our analysis and highlight some of the challenges. First, in our diagrams, we have implicitly assumed a disjointness property between the regions described by edges, which allows us to perform precise updates on points-to edges (as a developer would likely do in an informal verification). This assumption is made explicit by utilizing separation logic [Reynolds 2002] to formalize these diagrams (see Section 3.3), and which imposes an implicit linearity restriction on the checkers. Specifically, a checker must be a linear traversal over the data structure; in terms of dynamic checking, a compilation of a checker must check that each address is dereferenced at most once during the traversal (see Nguyen et al. [2008] for one method). Second, as with many data structure operations, the rebalance routine requires a traversal using a *cursor* (e.g., `c`). To check properties of such operations, we are often required to track information in detail locally around the cursor, but we may be able to summarize the rest rather coarsely. This summarization cannot only be for the suffix (yet to be visited by the cursor) but must also be for the prefix (already visited by the cursor) (see Section 3.3.1).

Some challenges are less apparent from the user's perspective but arise from taking the end-user approach and utilizing user-supplied data structure invariant checkers as specifications in static analysis. I sum up the challenges here, but I revisit them in more detail in Chapter 3 and Chapter 4. First, while we take advantage of the fact that user-supplied invariant checkers describe to the analysis one common way to use the data structure, it does not necessarily give sufficient information when the data structure invariants are richer (e.g., have back pointers, include data properties across elements). Second, similar to other shape analyses, a central challenge is to *summarize* the graphs sufficiently in order to find a fixed point while retaining enough precision. Our analysis summarizes by *folding* back into checker edges, but with arbitrary data structure specifications, it becomes particularly difficult. The key observation we make is that previous iterates are generally more abstract

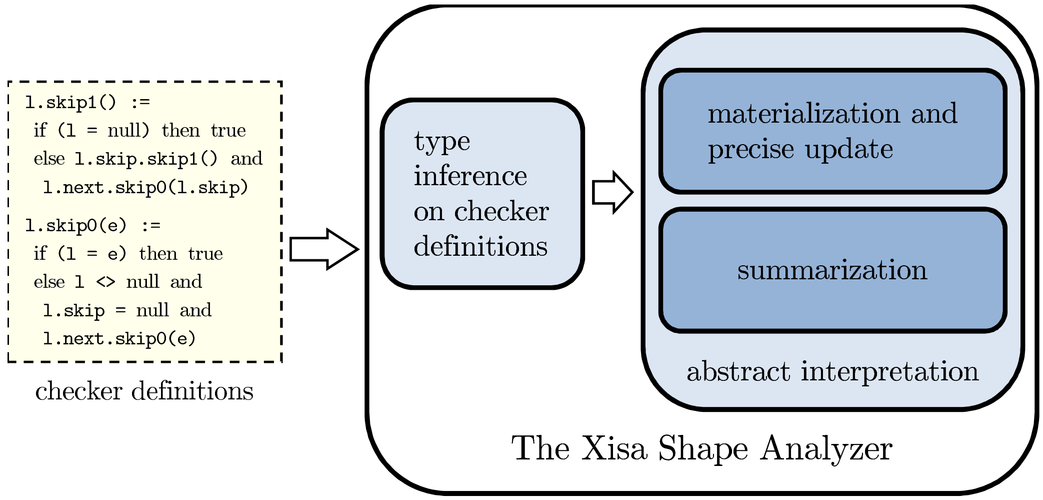


Figure 2.6: The basic architecture of Xisa.

and can be used to guide the folding process.

As depicted in Figure 2.6, our shape analysis tool, Xisa, takes as input, definitions of data structure invariant checkers and in the end, performs an abstract interpretation [Cousot and Cousot 1977] using the key shape analysis operations: materialization, precise update, and summarization (as described in Section 2.1). A novel aspect of our architecture is that in between, we first do a separate analysis on the checker definitions before analyzing the program text. It is in this phase that we derive additional information about the data structure that makes our analyzer more robust with respect to how checkers are defined. Speaking informally, it is this phase that takes testing code and makes it more amenable to being a specification for static analysis.

In short, the contributions of this work are as follows:

- We observe that data structure invariant checking code can help guide a shape analysis and provides a familiar mechanism for the developer to supply information to the analyzer. Intuitively, checkers can be viewed as developer-supplied summaries of heap regions bundled with a usage pattern for such regions (Section 3.2).

- We develop an end-user shape analysis whose shape abstraction is built from the user-supplied invariant checkers. Our abstract domain is not only parameterized by user-supplied invariant checkers, but also by a numerical domain for data constraints (Section 3.3).

- We introduce a notion of partial checker runs as part of the abstraction in order to generalize user-supplied summaries when the data structure invariant holds only partially (Section 3.3.1).

- We make our analysis more robust with respect to user specification by deriving additional information through a separate type analysis on the checker definitions in order to guide the abstract interpretation (Section 3.4). This additional information is particularly critical for "invertible structures," such as doubly-linked lists and trees with parent pointers. A novel aspect of our work is that "backward unfolding" operations for such structures are derived automatically from the standard forward unfolding (Section 4.1.1).

- We notice that the iteration history of the analysis can be used to guide the weakening of shape invariants (which could potentially apply to other shape analyses). We develop an automatic summarization strategy using a binary *widening* operator for our abstraction based on this observation. Also, we show that widening requires careful coordination between shape and data (Section 4.2).

Chapter 3

End-User Abstraction with Invariant Checkers

Chapter 2 motivates end-user shape analysis from the user's perspective. This chapter and the two subsequent chapters describe the design and implementation of our end-user shape analyzer in order to meet those expectations. In this chapter, we focus on our end-user shape abstraction using data structure invariant checkers, while Chapter 4 details our shape analysis algorithm using that memory abstraction. Finally, in Chapter 5, I present a discussion of our experience applying our analyzer.

As alluded to in Chapter 2, I identify two significant challenges from the analyzer design perspective:

1. how to be robust with respect to the way the user defines the invariant checkers; and

2. how to summarize memory states using invariant checkers (i.e., the user-supplied summarization mechanism) sufficiently to find a fixed point while retaining enough

precision to capture the properties of interest.

As mentioned there, these difficulties are only exacerbated by richer, *relational* data structure invariants, such as structures with back pointers (e.g., doubly-linked lists) and those that have data constraints across elements (e.g., sortedness) (as compared to the simpler, non-relational data structure invariants used in Chapter 2). Thus, before discussing the design and implementation of our end-user shape analyzer, we look more closely at relational invariants to better understand the challenges they pose on our analysis design (Section 3.1). Then, in Section 3.2, we discuss further the kinds of invariant checkers used by our analysis. Section 3.3 formalizes our memory abstraction and provides a connection between the executable interpretation of invariant checkers and how they are utilized to summarize memory regions in our analysis abstraction. Finally, the type analysis on checker definitions that is a key piece in making the analysis robust in the presence of user specification is defined in Section 3.4.

3.1 Challenge: Relational Invariants

Most shape analyses are very effective when the analysis can be done non-relationally, that is, the property of interest can be decomposed so that the checking of one part is (mostly) independent of the checking of others. A significant challenge for almost all shape analyses is to step beyond non-relational abstractions[1]. For our analysis, we will see that relational invariant checkers emphasize the need for deriving additional information about the specified data structures to guide the abstract interpretation.

[1]Our papers that describe parts of this work are split mostly along this divide between non-relational [Chang et al. 2007] and relational [Chang and Rival 2008] end-user shape analysis.

One of the simplest examples of a relational checker is the following definition for a doubly-linked list:

Example 3.1 (A doubly-linked list checker definition).

$$
\begin{aligned}
\texttt{l.dll}(\texttt{l}_{\mathrm{p}}) \;\; := \;\; & \textsf{if } (\texttt{l = null}) \textsf{ then true} \\
& \textsf{else } \texttt{l.prev = l}_{\mathrm{p}} \textsf{ and } \texttt{l.next.dll(l)}
\end{aligned}
$$

In the above, the `dll` checker says a doubly-linked list is either empty or has `next` and `prev` fields where `prev` must contain $\texttt{l}_{\mathrm{p}}$ and `next` must be a doubly-linked list whose `prev` is the current root pointer `l`. At a high-level, a non-relational checker describes each segment of the data structure independently, which includes the aforementioned checkers for singly-linked lists (Example 2.1) and skip lists (Figure 2.4(b)) but not the `dll` checker. Syntactically, in a non-relational checker, the additional parameters of the checker (i.e., the state of the checker) are constant across all recursive calls. This condition clearly holds for a stateless checker like the one for singly-linked lists. In contrast, the `dll` checker uses an additional parameter $\texttt{l}_{\mathrm{p}}$ to specify that the `prev` field of the next element points to the current element.

The main difficulty with relational checkers is that simply unfolding instances into their definitions does not reveal these relations. Thus, code that utilizes such relations is not analyzable without other techniques. With the `dll` checker, an unfolding reveals that `next` points to another `dll`, but not that `prev` must also point to a segment of `dll`, nor that `next` and `prev` are inverses (i.e., following `next` and then `prev` gets back to the same node). As such, analyzing code that traverses a doubly-linked list using the `next` field with the `dll` checker is easier than analyzing code that traverses using the `prev` field. Part of the problem is that there are a number of ways to traverse a doubly-linked list (i.e., a number of inductive schemes). For example, an alternative checker could start at the tail of the list following `prev` fields, but then the above difficulty is simply reversed for the fields.

The relational issue becomes even more salient when we consider inductive invariants with more involved data constraints than pointer equality. For example, consider the following checkers:

Example 3.2 (A binary search tree checker definition)**.**

$$\begin{array}{l}\texttt{t.bst}(\texttt{t}_{\mathrm{lo}},\texttt{t}_{\mathrm{up}}) \ \ := \\ \quad \text{if } (\texttt{t = null}) \text{ then true} \\ \quad \text{else } \texttt{t}_{\mathrm{lo}} < \texttt{t.d} < \texttt{t}_{\mathrm{up}} \\ \qquad\quad \text{and } \texttt{t.l.bst}(\texttt{t}_{\mathrm{lo}},\texttt{t.d}) \text{ and } \texttt{t.r.bst}(\texttt{t.d},\texttt{t}_{\mathrm{up}})\end{array}$$

Example 3.3 (A list of a given length checker definition)**.**

$$\begin{array}{l}\texttt{l.listn}(\texttt{l}_{\mathrm{len}}) \ \ := \\ \quad \text{if } (\texttt{l = null}) \text{ then } \texttt{l}_{\mathrm{len}} = 0 \\ \quad \text{else } \texttt{l.next.listn}(\texttt{l}_{\mathrm{len}} - 1)\end{array}$$

Each of these relational checkers uses parameters to capture very different kinds of relations and thus pose different challenges. The `bst` checker enforces a global ordering property on the data fields (`d`) with the range narrowing in recursive calls (*the shape constrains the data*), while `listn` uses $\texttt{l}_{\mathrm{len}}$ to specify the recursion depth (*the data constrains the shape*). Finally, the `dll` checker describes a kind of *invertible structural invariant* with a data structure that points to previous roots.

To see how relational invariant checkers pose challenges in analyzer design, we consider an example analysis of red-black tree insertion. I first describe informally the invariants of the red-black tree we consider here.

Inverse Invariant (cf., doubly-linked lists):

- The `p` field is the inverse of the `l` and `r` fields (i.e., for each node `n`, if `n->l` $\neq$ null, then `n->l->p` $=$ `n` and analogously for the `r` field).

Order Invariant (cf., binary search trees):

- For each node `n`, the values in the left subtree (i.e., the value of the `d` fields in the nodes reachable from `n->l`) are less than `n->d`, and the values of the right subtree are greater than `n->d`.

Balance Invariant (cf., lists of a given length):

- A node is either red or black (given by the `clr` field); null is considered black.
- The root is black.
- Both children of a red node are black.
- Every simple path (i.e., following `l` and `r` fields) from a node to a leaf contains the same number of black nodes.

Observe that each of these invariants describes a relation between nodes, and in the case of the order and balance invariants, they describe a global relation on all the nodes of the tree. To write code that checks this invariant, an invariant checker needs to carry some state (in order to turn the global relations specified above into local checks). Figure 3.1 presents a checker definition for red-black trees. We can see that the kinds of relations used by the `rbtree` checker have parallels to each of the relational checkers presented above. The $\mathtt{t}_{\mathrm{p}}$ parameter is analogous to the constraint on `prev` field in the `dll` checker; $\mathtt{t}_{\mathrm{lo}}$ and $\mathtt{t}_{\mathrm{up}}$ is as in the `bst` checker; and $\mathtt{t}_{\mathrm{bh}}$ is similar to the length parameter in the `listn` checker. As such, the red-black tree insertion routine shown in Figure 3.2 is a fairly representative example of the kinds of challenges in designing our end-user shape analyzer based on invariant checkers. These challenges appear in both the unfolding to materialize needed points-to edges and the folding to retain sufficient precision in the data constraints.

In Figure 3.2, I show the abstract memory state of the analysis at a number of program points. To keep the diagrams compact, I draw points-to edges only for the pointer fields, and when necessary, I notate the values of data fields above the node (e.g., at program point 7). For data constraints, our memory abstraction is parameterized by a *base abstract domain* whose coordinates (i.e., variables) are the symbolic values. In the examples, I note the data constraints that are necessary to get the desired results and assume the base domain can capture them. Also, I have elided the additional parameters on the instances of rbtree. Instead, I adopt the convention of referring to the additional parameters by subscripting the node name on which the checker applies. For example, the checker edge on line 1 conveys $\alpha.\mathsf{rbtree}(\alpha_{\mathrm{p}}, \alpha_{\mathrm{lo}}, \alpha_{\mathrm{up}}, \alpha_{\mathrm{redok}}, \alpha_{\mathrm{bh}})$ where any constraints on the additional parameters are given in the data domain.

Let us look at where the relational aspect of the `rbtree` checker poses obstacles to overcome in the analysis example. First, consider the memory state at program point 21 in the first iteration. Note that the value of `pa` is γ, which has no outgoing edges from it (neither points-to nor checker). However, on the next iteration, we analyze statements that access the fields of `pa` and thus need to materialize them. From our intuitive understanding of the `rbtree` checker, we know that γ lies on the segment between α and β (when it is non-empty), so if we were to unfold the segment *backward* from β, we could materialize the fields of γ. To justify the unfolding of partial checker edges, we have particular logical representation of such segments (see Section 3.3.2). The novel aspect of our proposal is that we determine when to apply this backward unfolding automatically using a separate type

```
typedef
  struct RBNode {
    int d; color clr;              // data, color
    struct RBNode *l, *r, *p;      // left child, right child, parent
  }
  RBNode;
```

(a) A C type definition for a red-black tree node.

$\mathtt{t.rbtree}(\mathtt{t}_{\mathrm{p}}, \mathtt{t}_{\mathrm{lo}}, \mathtt{t}_{\mathrm{up}}, \mathtt{t}_{\mathrm{redok}}, \mathtt{t}_{\mathrm{bh}})$:=

  if ($\mathtt{t}$ = null) then $\mathtt{t}_{\mathrm{bh}} = 0$

  else $\mathtt{t.p} = \mathtt{t}_{\mathrm{p}}$ and $\mathtt{t}_{\mathrm{lo}} < \mathtt{t.d} < \mathtt{t}_{\mathrm{up}}$ and ($\mathtt{t.clr} <> \mathsf{red}$ or $\mathtt{t}_{\mathrm{redok}}$) and

    $\mathtt{t.l.rbtree}(\mathtt{t}, \mathtt{t}_{\mathrm{lo}}, \mathtt{t.d}, \mathtt{t.clr} <> \mathsf{red}, \mathsf{ite}(\mathtt{t.clr} = \mathsf{red}, \mathtt{t}_{\mathrm{bh}}, \mathtt{t}_{\mathrm{bh}} - 1))$ and

    $\mathtt{t.r.rbtree}(\mathtt{t}, \mathtt{t.d}, \mathtt{t}_{\mathrm{up}}, \mathtt{t.clr} <> \mathsf{red}, \mathsf{ite}(\mathtt{t.clr} = \mathsf{red}, \mathtt{t}_{\mathrm{bh}}, \mathtt{t}_{\mathrm{bh}} - 1))$

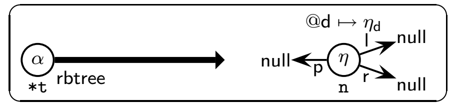
(b) A red-black tree checker definition.

Figure 3.1: Describing a red-black tree. In (b), the additional parameters to the `rbtree` checker are used to impose the following constraints:

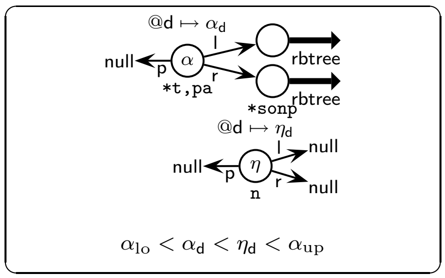
$\mathtt{t}_{\mathrm{p}}$	is where the `p` field should point;
$\mathtt{t}_{\mathrm{lo}}$	is a lower bound on the `d` field;
$\mathtt{t}_{\mathrm{up}}$	is an upper bound on the `d` field;
$\mathtt{t}_{\mathrm{redok}}$	gives whether the `clr` field is allowed to be red; and
$\mathtt{t}_{\mathrm{bh}}$	is the number of black nodes on all paths to leaves (i.e., the black height).

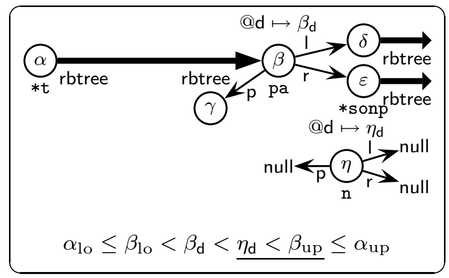
An ite is an if-then-else expression.

```
void insert(RBNode **t, RBNode *n) {
  RBNode *pa, **sonp, *son;
```

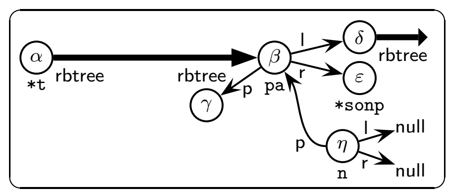



```
  pa = null; sonp = t;
  while (*sonp != null) {
    pa = *sonp;
    if      (n->d < pa->d) sonp = &(pa->l);
    else if (n->d > pa->d) sonp = &(pa->r);
    else                   return;
```

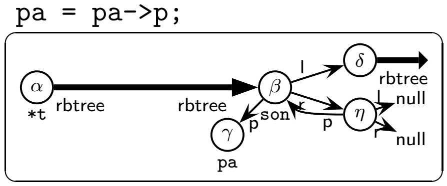


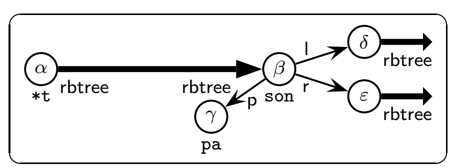



```
  }
  n->p = pa; n->clr = RED;
```




```
  *sonp = n; son = n;
  while (pa != null) {
    if (pa does not locally satisfy the red-black invariants) {
      ... perform rotations to reestablish invariants ...
    }
    son = pa; pa = pa->p;
```






```
  }
  son->clr = BLACK; *t = son;
}
```

first iteration | *at fixed point*

Figure 3.2: Red-black tree insertion in C.

inference on checker definitions (see Section 3.4). We also reduce the soundness justification of backward unfolding to that of forward unfolding (see Section 4.1.1). In other words, while the user supplies the forward unfolding axiom (as an inductive checker definition), the backward unfolding is derived automatically and not axiomatized (cf., Berdine et al. [2007]).

Second, consider the memory states at program point 7, which is a location where the combination of shape and relational data constraints makes analysis difficult. We get the data constraints in the first iteration simply by unfolding `rbtree` and from the guard on the conditional. The challenge is on widening to generate the fixed-point invariant. In this example, we need the underlined constraint to know that the insertion preserves the ordering invariant, but it is not necessarily easy to obtain. At a high-level, the core of the difficulty is that while the top of the tree can be fairly easily summarized into the segment from α to β, the data invariant we desire requires synthesizing relations between the prefix segment (α to β) and the suffix (from β). To address this difficulty, we make some observations that allow us to apply of some standard analysis techniques in this context. One, we observe that because the coordinates of the data domain are given by heap nodes, the data domain is rather sensitive to the large changes that result from widening in the shape domain. Thus, we delay widening elements of the data domain until the shape portion has converged (keeping symbolic joins instead). Two, we notice that we can separate this synthesis task into finding appropriate arguments for the checker parameters and inferring the appropriate relation on those arguments. Specifically, we can make the finding of checker arguments shape-guided by using additional data fields that correspond to the checker parameters.

After describing our abstraction (this chapter) and our analysis algorithm (Chapter 4), we return to the red-black tree insertion example in Section 5.1 to discuss how our algorithm can be used to verify such operations.

3.2 Data Structure Invariant Checkers

Thus far, we have built up some intuition on how data structure invariant checkers describe particular shapes and data constraints. In this section, I detail the kinds of checkers we consider by defining a language of checker definitions suitable for input to our analysis.

3.2.1 Shapes Expressible as Checkers

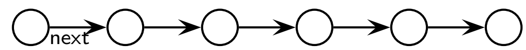
Intuitively, our view is that checkers convey shapes implicitly by the sequence of dereferences (i.e., field reads) they would make during the course of a successful dynamic execution. These dereferences are used to generate points-to edges for the analysis during materialization. Because we restrict edges to describe disjoint regions, these dereferences also need to be mutually distinct. In particular, one of the more constraining restrictions on checkers is that they must visit the data structure linearly—dereferencing an object-field at most once during the course of the traversal. It is easy to work around this restriction locally within a recursive call by first binding the value of a dereference into a new local variable (cf., the `skip1` checker of Figure 2.4(b) or the `bst` checker of Example 3.2).

Additionally, we consider checkers that are side-effect free and written in a recursive style. The user could potentially write in an iterative style as long as the traversal can be translated into a recursive one. As the notation suggests, we also distinguish a traversal

Checker Definition	Concrete Instance

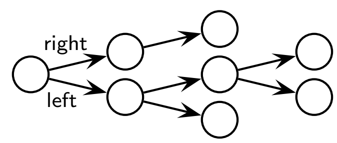
singly-linked lists

```
l.list() :=
  if (l = null) then true
  else l.next.list()
```



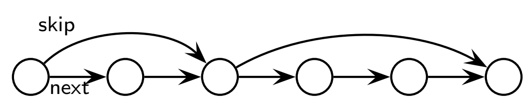
binary trees

```
t.tree() :=
  if (t = null) then true
  else t.left.tree() and
       t.right.tree()
```



two-level skip lists

```
l.skip1() :=
  if (l = null) then true
  else l.skip.skip1() and
       l.next.skip0(l.skip)
l.skip0(l_end) :=
  if (l = l_end) then true
  else l.skip = null and
       l.next.skip0(l_end)
```



cyclic lists

```
l.clist() :=
  if (l = null) then true
  else l.next.ls(l)
l.ls(l_end) :=
  if (l = l_end) then true
  else l.next.ls(l_end)
```

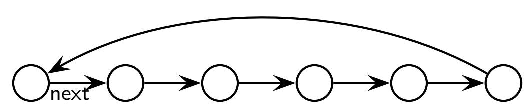
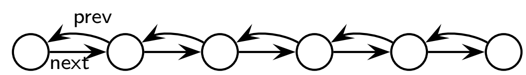


Table 3.1: Representative examples of shapes expressible with data structure invariant checkers (non-relational shapes). The graphs in the right column show data structure instances described by the checkers in the left column. For compactness, edges that point to null are not drawn, and field names are labeled only once.

Checker Definition	Concrete Instance
doubly-linked lists	

`l.dll(l_p) :=`
`  if (l = null) then true`
`  else l.prev = l_p and`
`       l.next.dll(l)`
	

binary trees with parent pointers	

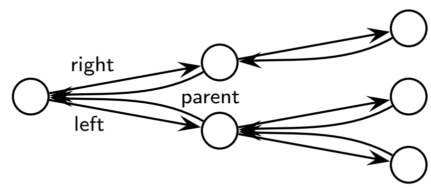
`t.dltree(t_p) :=`
`  if (t = null) then true`
`  else l.parent = t_p and`
`       t.left.dltree(t) and`
`       t.right.dltree(t)`
	

Table 3.2: Representative examples of shapes expressible with data structure invariant checkers (relational shapes).

parameter used to walk the data structure. A distinguished traversal parameter is not a strict requirement for our analysis (see Section 3.4), but as it is the common case, it makes the checker edges in the graph notation suggestive of a traversal and is assistive in the widening algorithm.

Despite these restrictions, we can still capture a rather wide range of shapes and covering most of those typically considered in shape analysis. In Table 3.1 and Table 3.2, I show some representative examples of shapes that can be expressed with checkers (collecting some of the previous examples). The examples are split into two parts, dividing the non-relational checkers (Table 3.1) from the relational ones (Table 3.2). They are roughly ordered from least to most complex. I focus only on shapes here, but note that data constraints add additional (but mostly orthogonal) dimensions of complexity (e.g., `bst` of Example 3.2 where the shape constrains the data or `listn` of Example 3.3 where the data constrains the shape).

Arbitrary directed-acyclic graphs (DAGs) are, however, not expressible in this language, though special cases, like skip lists, are. From Table 3.1 and Table 3.2, we can see that the additional parameters provide a way to express "regular sharing patterns", but to traverse a DAG linearly, a checker would need to either keep a set of nodes it has already visited or have a way to mark visited nodes, which is outside the class of checkers we consider.

3.2.2 A Language of Checker Definitions

Figure 3.3(a) presents a language of validation expressions that we have seen in example checker definitions (as in Table 3.1 and Table 3.2). For presentation, I write checkers with one traversal parameter and one additional parameter $\pi\texttt{.}c(\rho)$ (where π is the traversal parameter and ρ is the additional parameter), though everything applies to checkers with zero-or-more additional parameters. Also, to make it easier to describe certain aspects, I introduce a hierarchy in the expressions. Conceptually, validation expressions *ve* describe possible memory states, validation shapes *vs* express a single memory state, and validation terms *vt* give Boolean constraints. Validation terms *vt* and validation values u can be extended with whatever relations and constants of interest. Finally, an access path *ap* is simply a sequence of field reads.

Because field reads are at the core of our notion of shape, it is easier to work with checkers defined in a normal form where a let-binding is introduced for each read and each use of that value is replaced by reference to the newly introduced name. In particular, this normalization turns multiple reads through the same access path into the binding of a

$$
\begin{array}{llcl}
\text{checker definitions} & \mathit{vchkdef} & ::= & \pi\,.\,c(\rho)\ \texttt{:=}\ \mathit{ve} \\[1em]
\text{validation expressions} & \mathit{ve} & ::= & \mathit{vs} \mid \textsf{if}\ \mathit{vt}\ \textsf{then}\ \mathit{ve}_0\ \textsf{else}\ \mathit{ve}_1 \\
\text{validation shapes} & \mathit{vs} & ::= & \mathit{vt} \mid \mathit{vt}_0\,.\,c(\mathit{vt}_1) \mid \mathit{vs}_0\ \textsf{and}\ \mathit{vs}_1 \\
\text{validation terms} & \mathit{vt} & ::= & u \mid \mathit{ap} \mid \mathit{vt}_0\ \texttt{=}\ \mathit{vt}_1 \mid \cdots \\[1em]
\text{values} & u & ::= & \textsf{true} \mid \textsf{false} \mid \textsf{null} \mid \cdots \\[1em]
\text{access paths} & \mathit{ap} & ::= & \alpha \mid \mathit{ap}\,.\,f \\[1em]
\text{symbolic values} & \alpha & \in & \mathbf{Val}^\sharp \\
\text{field names} & f, g & & \\
\text{checker names} & c & &
\end{array}
$$

(a) A language of validation expressions.

$$
\begin{array}{llcl}
\text{checker definitions} & \mathit{chkdef} & ::= & \pi\,.\,c(\rho)\ \texttt{:=}\ e \\[1em]
\text{validation expressions} & e & ::= & s \mid \textsf{if}\ t\ \textsf{then}\ e_0\ \textsf{else}\ e_1 \mid \textsf{let}\ \beta\ \texttt{=}\ \alpha\,.\,f\ \textsf{in}\ e \\
\text{validation shapes} & s & ::= & t \mid \alpha\,.\,c(\delta) \mid s_0\ \textsf{and}\ s_1 \mid \textsf{let}\ \alpha\ \texttt{=}\ t\ \textsf{in}\ s \\
\text{validation terms} & t & ::= & u \mid \alpha \mid t_0\ \texttt{=}\ t_1 \mid \cdots
\end{array}
$$

(b) A language of validation expressions with normalized memory reads.

Figure 3.3: Abstract syntax of the checker definition language.

single read followed by uses of that binding. Figure 3.3(b) shows a modified version of the checker definition language introducing a **let** construct for field reads and discarding the use of access paths. Simply for convenience, we also restrict calls to apply to symbolic values by introducing other **let** bindings. We can syntactically translate between the languages as discussed below.

Normalization

Figure 3.4(a) shows a normalization of ve by introducing let-bindings for each field read. I write L for a sequence of let-bindings

$$\beta_0 = \alpha_0 \,.\, f_0, \beta_1 = \alpha_1 \,.\, f_1, \ldots, \beta_n = \alpha_n \,.\, f_n$$

and maintain the invariant that each β_i and each $\alpha_i \,.\, f_i$ are distinct. Also, I write **let** L **in** e to mean the expression given by a sequence of **let**s followed by e. The rules in the top line describe the introduction of the let-bindings for each distinct field read, while the next group of rules simply propagates the bindings to the expression level. We consider the let-bindings to always introduce new names (i.e., as a side-condition, variables in the inputs ve, vs, vt, and ap are never captured by L). At the expression level, we introduce **let** expressions for the newly introduced bindings (i.e., L'). Note that we consider the field reads of each branch of an **if-then-else** separately (i.e., introduce separate **let** expressions in each branch).

I show below examples of normalized checker definitions for two-level skip lists and doubly-linked lists.

Example 3.4 (A normalized two-level skip list checker)**.** Here, we have a normalized def-

$ve\ ;\ L \Rsh\ e$	ve with bindings L compiles to e.
$vs\ ;\ L \Rsh\ L' \ \mathsf{in}\ s$	vs with bindings L compiles to s with L'.
$vt\ ;\ L \Rsh\ L' \ \mathsf{in}\ t$	vt with bindings L compiles to t with L'.
$ap\ ;\ L \Rsh\ L' \ \mathsf{in}\ \alpha$	ap with bindings L compiles to α with L'.

$$\frac{(\alpha \text{ not captured by } L)}{\alpha\ ;\ L \Rsh\ L\ \mathsf{in}\ \alpha} \qquad \frac{ap\ ;\ L \Rsh\ L'\ \mathsf{in}\ \alpha \quad (\beta = \alpha\,.\,f \text{ in } L')}{ap\,.\,f\ ;\ L \Rsh\ L'\ \mathsf{in}\ \beta} \qquad \frac{ap\ ;\ L \Rsh\ L'\ \mathsf{in}\ \alpha \quad (\beta \text{ fresh})}{ap\,.\,f\ ;\ L \Rsh\ L', \beta = \alpha\,.\,f\ \mathsf{in}\ \beta}$$

$$\frac{}{u\ ;\ L \Rsh\ L\ \mathsf{in}\ u} \qquad \frac{vt_0\ ;\ L \Rsh\ L'\ \mathsf{in}\ t_0 \quad vt_1\ ;\ L' \Rsh\ L''\ \mathsf{in}\ vt_1}{vt_0 = vt_1\ ;\ L \Rsh\ L''\ \mathsf{in}\ t_0 = t_1}$$

$$\frac{vt_0\ ;\ L \Rsh\ L'\ \mathsf{in}\ t_0 \quad vt_1\ ;\ L' \Rsh\ L''\ \mathsf{in}\ t_1}{vt_0\,.\,c(vt_1)\ ;\ L \Rsh\ L''\ \mathsf{in}\ \mathsf{let}\ \alpha = t_0\ \mathsf{in}\ \mathsf{let}\ \delta = t_1\ \mathsf{in}\ \alpha\,.\,c(\delta)}$$

$$\frac{vs_0\ ;\ L \Rsh\ L'\ \mathsf{in}\ s_0 \quad vs_1\ ;\ L' \Rsh\ L''\ \mathsf{in}\ vs_1}{vs_0\ \mathsf{and}\ vs_1\ ;\ L \Rsh\ L''\ \mathsf{in}\ s_0\ \mathsf{and}\ s_1}$$

$$\frac{vs\ ;\ L \Rsh\ L, L'\ \mathsf{in}\ t}{vs\ ;\ L \Rsh\ \mathsf{let}\ L'\ \mathsf{in}\ t} \qquad \frac{vt\ ;\ L \Rsh\ L, L'\ \mathsf{in}\ t \quad ve_0\ ;\ L, L' \Rsh\ e_0 \quad ve_1\ ;\ L, L' \Rsh\ e_1}{\mathsf{if}\ vt\ \mathsf{then}\ ve_0\ \mathsf{else}\ ve_1\ ;\ L \Rsh\ \mathsf{let}\ L'\ \mathsf{in}\ \mathsf{if}\ t\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1}$$

$$\begin{array}{rcl} \Rsh(ve) \overset{\text{def}}{=} e & \text{if} & ve\ ;\ \cdot \Rsh\ e \\ \Rsh(\pi\,.\,c(\rho)\ := ve) & \overset{\text{def}}{=} & \pi\,.\,c(\rho)\ := \Rsh(ve) \end{array}$$

(a) Compilation into normalized expressions.

$\Lsh(e) \overset{\text{def}}{=} ve$	e decompiles to ve.

$$\begin{array}{rcl} \Lsh(\mathsf{let}\ \beta = \alpha\,.\,f\ \mathsf{in}\ e) & \overset{\text{def}}{=} & [\alpha\,.\,f/\beta]\Lsh(e) \\ \Lsh(\mathsf{let}\ \alpha = t\ \mathsf{in}\ s) & \overset{\text{def}}{=} & [t/\alpha]\Lsh(s) \end{array}$$

$$\begin{array}{rcl@{\qquad}rcl} \Lsh(u) & \overset{\text{def}}{=} & u & \Lsh(\alpha\,.\,c(\delta)) & \overset{\text{def}}{=} & \alpha\,.\,c(\delta) \\ \Lsh(\alpha) & \overset{\text{def}}{=} & \alpha & \Lsh(s_0\ \mathsf{and}\ s_1) & \overset{\text{def}}{=} & \Lsh(s_0)\ \mathsf{and}\ \Lsh(s_1) \\ \Lsh(t_0 = t_1) & \overset{\text{def}}{=} & \Lsh(t_0) = \Lsh(t_1) & \Lsh(\mathsf{if}\ t\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \overset{\text{def}}{=} & \mathsf{if}\ \Lsh(t)\ \mathsf{then}\ \Lsh(e_0)\ \mathsf{else}\ \Lsh(e_1) \end{array}$$

(b) Decompilation of normalized expressions.

Figure 3.4: Translation between validation expressions.

inition of the two-level skip list checker from Figure 2.4(b).

```
l.skip1()      :=  if (l = null) then true
                   else
                     let s = l.skip in
                     let n = l.next in
                     s.skip1() and n.skip0(s)

l.skip0(end)   :=  if (l = end) then true
                   else
                     let s = l.skip in
                     let n = l.next in
                     s = null and n.skip0(end)
```

Example 3.5 (A normalized doubly-linked list checker definition)**.** Below is a normalized definition of the doubly-linked list checker from Example 3.1.

```
l.dll(l_p)  :=  if (l = null) then true
                else
                  let p = l.prev in
                  let n = l.next in
                  p = l_p and n.dll(l)
```

To recover the original version, Figure 3.4(b) shows a simple inverse translation defined by induction over the structure of expressions, which simply eliminates **let**s by performing substitutions. I write $[\alpha\,.\,f/\beta]ve$ for substituting $\alpha\,.\,f$ for β in ve (and similarly for $[vt/\alpha]vs$). We can state that these translations are indeed inverses. I write $[L]ve$ for iteratively substituting with the bindings of L in ve.

Theorem 3.1 (Recovering normalized expressions)**.**

1. If $ve\ ;\ L \Rsh\ e$, *then* $[L]\Lsh(e) = ve$.

2. *If* $vs\ ; L \Rsh L'$ in s, *then* $L' \supseteq L$ *and* $[L']^{\Lsh}(s) = vs$.

3. *If* $vt\ ; L \Rsh L'$ in t, *then* $L' \supseteq L$ *and* $[L']^{\Lsh}(t) = vt$.

4. *If* $ap\ ; L \Rsh L'$ in α, *then* $L' \supseteq L$ *and* $[L']^{\Lsh}(\alpha) = ap$.

Proof. By induction on the given derivations (appealing to the subsequent fact in the proof of each one). □

Evaluation Semantics

The goal of the normalization is to define an evaluation semantics of the validation expressions that characterizes how checkers convey shape. To do so, I write $u \in \mathbf{Val}$ for the set of values. Furthermore, I write $u + f$ for the base address u plus the offset of field f. A concrete store $\sigma\colon \mathbf{Val} \rightharpoonup_{\text{fin}} \mathbf{Val}$ maps addresses into values where we make no distinction between addresses and values of the store. I write $[\cdot]$ for the empty concrete store and $[u_0 \mapsto u_1]$ for the store of one cell mapping u_0 to u_1. A compound store $\sigma_0 * \sigma_1$ is a store with disjoint substores σ_0 and σ_1 where σ_0 and σ_1 must have disjoint domains. To assign an interpretation to symbolic values, we consider a *valuation* ν that is a mapping from symbolic values into concrete values.

Intuitively, we view checkers as "consuming" the memory cells that it dereferences during its traversal to ensure that it visits the data structure linearly. The cells that it consumes are then the cells that make up the data structure. The evaluation judgment

$$\nu \vdash \langle \sigma, e \rangle \Downarrow \langle \sigma', u \rangle \ ,$$

reflects this view by producing both a residual store σ' and a value u. It says, "Under valuation ν and in store σ, expression e leaves a residual store σ' and evaluates to value

$\nu \vdash \langle \sigma, e \rangle \Downarrow \langle \sigma', u \rangle$	Under valuation ν and in store σ, expression e leaves a residual store σ' and evaluates to value u.
$\nu \vdash t \Downarrow u$	Under valuation ν, term t evaluates to value u.

$$\frac{\nu, \beta \to u \vdash \langle \sigma, e \rangle \Downarrow \langle \sigma', u \rangle}{\nu \vdash \langle [\nu(\alpha) + f \mapsto u] * \sigma, \mathsf{let}\ \beta = \alpha\,.\,f\ \mathsf{in}\ e \rangle \Downarrow \langle \sigma', u \rangle}\ \textsf{e-read}$$

$$\frac{\nu \vdash t \Downarrow \mathsf{true} \quad \nu \vdash \langle \sigma, e_0 \rangle \Downarrow \langle \sigma', u \rangle}{\nu \vdash \langle \sigma, \mathsf{if}\ t\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1 \rangle \Downarrow \langle \sigma', u \rangle}\ \textsf{e-if-t} \qquad \frac{\nu \vdash t \Downarrow \mathsf{false} \quad \nu \vdash \langle \sigma, e_1 \rangle \Downarrow \langle \sigma', u \rangle}{\nu \vdash \langle \sigma, \mathsf{if}\ t\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1 \rangle \Downarrow \langle \sigma', u \rangle}\ \textsf{e-if-f}$$

$$\frac{\nu \vdash \langle \sigma, [\alpha, \delta/\pi, \rho] e \rangle \Downarrow \langle \sigma', u \rangle \quad (\pi\,.\,c(\rho)\ := e)}{\nu \vdash \langle \sigma, \alpha\,.\,c(\delta) \rangle \Downarrow \langle \sigma', u \rangle}\ \textsf{e-call}$$

$$\frac{\nu \vdash \langle \sigma, s_0 \rangle \Downarrow \langle \sigma', u_0 \rangle \quad \nu \vdash \langle \sigma', s_1 \rangle \Downarrow \langle \sigma'', u_1 \rangle}{\nu \vdash \langle \sigma, s_0\ \mathsf{and}\ s_1 \rangle \Downarrow \langle \sigma'', u_0 \wedge u_1 \rangle}\ \textsf{e-and}$$

$$\frac{\nu \vdash t \Downarrow u_0 \quad \nu, \alpha \to u_0 \vdash \langle \sigma, s \rangle \Downarrow \langle \sigma', u \rangle}{\nu \vdash \langle \sigma, \mathsf{let}\ \alpha = t\ \mathsf{in}\ s \rangle \Downarrow \langle \sigma', u \rangle}\ \textsf{e-let-term} \qquad \frac{\nu \vdash t \Downarrow u}{\nu \vdash \langle \sigma, t \rangle \Downarrow \langle \sigma, u \rangle}\ \textsf{e-term}$$

$$\frac{}{\nu \vdash u \Downarrow u}\ \textsf{e-val} \qquad \frac{}{\nu \vdash \alpha \Downarrow \nu(\alpha)}\ \textsf{e-var} \qquad \frac{\nu \vdash t_0 \Downarrow u_0 \quad \nu \vdash t_1 \Downarrow u_1}{\nu \vdash t_0 = t_1 \Downarrow u_0 = u_1}\ \textsf{e-eq}$$

Figure 3.5: Evaluation of validation expressions.

u." Figure 3.5 defines the evaluation judgment. The key rule is e-read, which is shown at the top. A read requires that the field exists in the store, and then upon reading, the cell is dropped from store. Its contents are saved by extending the valuation so that it can be used in evaluating the body of the let expression. With this view, a store is described by a checker evaluation if it is entirely consumed by a checker evaluation that yields the true value (i.e., $\sigma' = [\cdot]$ and $u = \mathsf{true}$) (see Section 3.3.3).

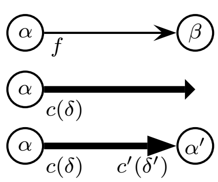
The remaining rules are mostly as expected. An if-then-else expression is evaluated by first evaluating the guard and choosing the appropriate branch (rules e-if-t and e-if-f). The normalization from Figure 3.4(a) ensures that only the fields used by a particular branch are consumed (but a field read in the guard applies to both branches). The e-call rule says that a checker call is evaluated by unrolling to its body. Finally, observe that because of the normalization, terms t are independent of the store.

3.3 Memory Abstraction

The core component of our analysis state is an abstract memory state M (as shown in Figure 3.6(a)). The memory state is based largely on separation logic [Reynolds 2002], so we use notation that is borrowed from there. No prior background in separation logic will be expected in this chapter; any needed information will be provided as necessary. The first three items are as in separation logic: emp is the empty memory, $\alpha@f \mapsto \beta$ is a points-to relation describing a single memory cell, and $M_0 * M_1$ is the separating conjunction specifying a memory that can be divided into two disjoint regions (i.e., with disjoint domains). A field offset expression $\alpha@f$ corresponds to the base address α plus the

memories	M	$::=$	emp	empty
		$\mid$	$\alpha@f \mapsto \beta$	memory cell
		$\mid$	$M_0 * M_1$	disjoint regions
		$\mid$	$\alpha.c(\delta)$	checker region
		$\mid$	$\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$	checker segment
data constraints	P	$\in$	$\mathbb{P}^\sharp$	
environments	E	$::=$	$\cdot \mid E, x \mapsto\!\!\!\!\rightarrow \alpha$	
analysis states	A	$::=$	$\bot \mid \langle E, M, P\rangle \mid A_1 \vee A_2$	
symbolic values	α	$\in$	$\mathbf{Val}^\sharp$	
program variables	x	$\in$	$\mathbf{Var}$	
field names	f, g			
checker names	c			

(a) The abstract memory state as formulas in separation logic.



	$\alpha@f \mapsto \beta$	memory cell
	$\alpha.c(\delta)$	checker region
	$\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$	checker segment

(b) The correspondence between graph edges and logical formulas.

Figure 3.6: The analysis state.

offset of field f (i.e., `&(a.f)` in C). In this core language, we limit address computation to field offsets (rather than arbitrary pointer arithmetic) and assume that all pointers occur as fields in a `struct`. Also for simplicity in presentation, we consider only symbolic values on the right side of points-to (β) (i.e., for the contents of memory cells) and assume any additional constraints on those values are captured elsewhere (e.g., is null). Together these components can symbolically describe a single, finite memory. The next two items are used to summarize memory regions. An application of a user-supplied checker $\alpha.c(\delta)$ describes a memory region where c succeeds when applied to α and δ. The last item provides a generic mechanism for specifying segments of user-supplied checkers, which I describe below in Section 3.3.1. In Figure 3.6(b), I show the correspondence between formulas and graphs. The thick edges (for checker regions and checker segments) can be intuitively thought of as representing possible subgraphs of thin points-to edges arranged in particular shapes and with certain constraints.

3.3.1 Checker-Based Summaries

In this subsection, we look in more detail how our abstraction is built out of user-supplied checker definitions. Recall that a dynamic run of a checker, such as the checker examples shown in Table 3.1 and Table 3.2, visits a region of memory starting from some root pointer, and furthermore, a successful, terminating run of a checker indicates how the user intends to access that region of memory. In the context of our analysis, a checker gives a corresponding inductively-defined predicate in separation logic and a successful, terminating run of the checker bears witness to a derivation of that predicate (see Section 3.3.3).

Inductive Checker Definitions

Formally, the definition of a checker c, with traversal parameter π and additional parameter ρ is a finite disjunction of checker rules. A checker rule R consists of a conjunction of a memory portion M and a pure portion F (given as a first-order formula). Further, M is separated into two parts: an unfolded region M^{u} given by a series of points-to edges and a folded region M^{f} given by a series of checker applications. Schematically, checker definitions are written as follows:

$$\text{checker definitions} \qquad \pi.c(\rho) \;:=\; \langle M_0^{\mathrm{u}} * M_0^{\mathrm{f}}, F_0 \rangle \vee \cdots \vee \langle M_n^{\mathrm{u}} * M_n^{\mathrm{f}}, F_n \rangle$$

Free variables in the rules are considered as existential variables bound at the definition. Note that because we view checkers as code, the kinds of inductive predicates are further restricted. In particular, M^{u} correspond to finite access paths from the parameters (from π primarily) and thus existentials are only for the values of fields along those access paths. Each checker call in M^{f} is applied to arguments among the parameters (the traversal or the additional ones) or the existentials introduced in M^{u}. These kinds of inductive definitions are apt for analysis, as unfolding such a definition corresponds to materializing points-to edges from π. Section 3.3.3 gives a translation from the validation expressions of Figure 3.3 (in Section 3.2.2) to definitions of this form.

Inductive Segments and Partial Checker Runs

Inductive predicates from checkers give us rather precise summaries of memory regions, but unfortunately, they are typically not general enough to capture the invariants of interest at all program points. For example, consider the fixed-point invariant at program point 7 in the red-black tree example given in Figure 3.2 (i.e., the loop invariant of the first loop). Here, we must track some information in detail around a cursor (e.g., `pa` and `*sonp`), while we need to summarize *both* the already explored prefix before the cursor and the yet to be explored suffix after the cursor. Such a situation is typical when analyzing a traversal algorithm. The suffix can be summarized by checker applications $\delta.\mathsf{rbtree}(...)$ and $\varepsilon.\mathsf{rbtree}(...)$ (i.e., the rbtree edges from δ and ε), but unfortunately, the prefix segment between α and β (i.e., the region between the root pointer `*t` and the cursor pointer `pa`) cannot.

Rather than require more general checker specifications sufficient to capture these intermediate invariants, we introduce a generic mechanism for summarizing prefix segments. We make the observation that such a segment region is captured by a *partial checker run* (i.e., conceptually, a non-failing execution of a checker up to a certain point in its traversal). Or, in terms of inductively-defined predicates, a segment region is a kind of partial derivation with a hole in a subtree.

A key aspect in defining segments is that in order to support materialization of segments, they must also have an inductive structure. This additional structure then enables the definition of forward and backward unfolding schemes on segments. Informally,

$$\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$$

is a memory region where it satisfies $\alpha.c(\delta)$ up to some number of unfoldings and conjoining any disjoint memory that satisfies $\alpha'.c'(\delta')$ makes the combined region satisfy $\alpha.c(\delta)$.

There have been others who have also observed the need to introduce a way to describe endpoints in order to summarize data structure fragments individually (e.g., the existentials in Gulwani and Tiwari [2007] and the truncation points of Guo et al. [2007]). An important feature of our definition is that it not only enables summarization but also materialization through forward and backward unfolding. In Section 3.3.2, we discuss this definition in further detail, and in Section 4.1, we look at and justify the unfolding operations on segments.

Analysis State

To track data constraints (i.e., non-shape constraints), we maintain a pure state P, which we assume is an element of an abstract domain $\mathbb{P}^\sharp$. Note that the base data domain $\mathbb{P}^\sharp$ is a parameter of the analysis. To connect the abstract memory with the program, we also keep an environment E that maps program variables to symbolic values that denote their addresses. Finally, the overall analysis state A is a finite disjunction of $\langle E, M, P\rangle$ tuples[2]. The number of disjuncts in the analysis state at any program point is usually quite small, as the checker-based summaries include a notion of merging disjunctive cases. Intuitively, user-supplied checker definitions provide an indication of the cases that can be merged for the properties of interest. The fact that our summaries can describe empty memory regions also helps keep the number of disjuncts down (similar to the TVLA optimization of Arnold [2006]).

3.3.2 Semantics of the Memory Abstraction

In this subsection, I give the semantics of our memory abstraction. I focus mostly on segments, as other aspects of the graph follow mostly from separation logic [Reynolds

[2] In implementation, we only need to keep one environment E per static scope, as we map variables to symbolic values that denote their addresses.

$$\langle [\cdot], \nu \rangle \models \mathsf{emp} \qquad \text{(always, for all } \nu\text{)}$$

$$\langle [\nu(\alpha) + f \mapsto \nu(\beta)], \nu \rangle \models \alpha @ f \mapsto \beta \qquad \text{(always, for all } \nu\text{)}$$

$$\langle \sigma_0 * \sigma_1, \nu \rangle \models M_0 * M_1 \quad \text{iff} \quad \langle \sigma_0, \nu \rangle \models M_0 \text{ and } \langle \sigma_1, \nu \rangle \models M_1$$

$$\langle \sigma, \nu \rangle \models \alpha.c(\delta) \quad \text{iff} \quad \text{there exists an } i \text{ s.t. } \langle \sigma, \nu \rangle \models \alpha.c^i(\delta)$$

$$\langle \sigma, \nu \rangle \models \alpha.c^{i+1}(\delta)$$

iff there exists a checker rule $\langle M^{\mathrm{u}} * M^{\mathrm{f}}, F \rangle$ in the definition of $\pi.c(\rho)$ where $M^{\mathrm{f}} = \beta_0.c_0(\gamma_0) * \cdots * \beta_m.c_m(\gamma_m)$ and such that

$$\nu \text{ satisfies } [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]F \quad \text{and}$$
$$\langle \sigma, \nu \rangle \models [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]M^{\mathrm{u}} * \beta_0.c_0^i(\gamma_0) * \cdots * \beta_m.c_m^i(\gamma_m)$$

where $\vec{\kappa}$ are the free variables of the rule and $\vec{\varepsilon}$ are fresh

$$\langle \sigma, \nu \rangle \models \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta') \quad \text{iff} \quad \text{there exists an } i \text{ s.t. } \langle \sigma, \nu \rangle \models \alpha.c(\delta) \mathrel{*\!\!=}^i \alpha'.c'(\delta') \tag{3.1a}$$

$$\langle [\cdot], \nu \rangle \models \alpha.c(\delta) \mathrel{*\!\!=}^0 \alpha'.c(\delta') \quad \text{iff} \quad \nu(\alpha) = \nu(\alpha') \text{ and } \nu(\delta) = \nu(\delta') \tag{3.1b}$$

$$\langle \sigma, \nu \rangle \models \alpha.c(\delta) \mathrel{*\!\!=}^{i+1} \alpha'.c'(\delta') \tag{3.1c}$$

iff there exists a checker rule $\langle M^{\mathrm{u}} * (M^{\mathrm{f}} * \beta.c''(\gamma)), F \rangle$ in the definition of $\pi.c(\rho)$ such that

$$\nu \text{ satisfies } [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]F \quad \text{and}$$
$$\langle \sigma, \nu \rangle \models [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]M^{\mathrm{u}} * M^{\mathrm{f}} * (\beta.c''(\gamma) \mathrel{*\!\!=}^i \alpha'.c'(\delta'))$$

where $\vec{\kappa}$ are the free variables of the rule and $\vec{\varepsilon}$ are fresh

Figure 3.7: The semantics of the memory abstraction.

2002]. Recall that a concrete store σ maps addresses to values and a valuation ν maps symbolic values into concrete values. The valuation ν is used to capture relations among memory cells. In Figure 3.7, I define $\langle\sigma,\nu\rangle \models M$ to mean a concrete store σ and a valuation ν satisfy an abstract memory M. I write $[\alpha/\pi]M$ for substituting α for π in M.

Checker Regions

The semantics of a checker application is defined by induction over the "height of the underlying calling tree". I write $\alpha.c^i(\delta)$ for a checker application of height at most i. Intuitively, a checker application of height 1 should make no recursive calls (i.e., it should correspond to a case where M^{f} is emp). Observe that the valuation ν is what connects regions and thus allows checkers to be relational.

Segment Regions

In a similar way, the semantics of a segment $\alpha.c(\delta) \ast\!\!= \alpha'.c'(\delta')$ is defined by induction over the number of checker rule applications needed to build a derivation of $\alpha.c(\delta)$ from a derivation of $\alpha'.c'(\delta')$. For this purpose, we add an index i on segments to indicate its length (when it is known). Thus, the standard segment is one of zero-or-more steps (3.1a). The definition of $\ast\!\!=^i$ then proceeds by induction on i. Intuitively, we want a 0-step segment to be an "empty partial derivation" of $\alpha.c(\delta)$, which means it should be case that the checkers and arguments match and correspond to an empty store (3.1b). In Section 4.1.1, we will see that these restrictions are critical for the backward unfolding (as used in the red-black tree example of Figure 3.2). The definition of the inductive case is very similar to the corresponding case for checker applications. For an $(i{+}1)$-step segment, we unfold the head checker c once materializing points-to edges and a series of recursive checker calls where one of these checker calls should be replaced with a segment of rank i (3.1c).

Concretization

Finally, I tie these definitions together by giving the concretization of our abstract domain, which we assume is a reduced product [Cousot and Cousot 1979] of the shape domain (of graphs) and the base data domain. The concretization of an abstract memory state M is simply the set of store-valuation pairs that satisfy M. As the concrete counterpart of the environment E, I write $\theta\colon \mathbf{Var} \to \mathbf{Val}$ for a *concrete environment* that maps variables to concrete values. Here, I overload the concretization operator $\boldsymbol{\gamma}$ to apply to each of the component parts.

Definition 3.2 (Concretization)**.**

$$\begin{aligned}
\boldsymbol{\gamma}(\, M\,) &= \{\, (\sigma,\nu) \mid \langle\sigma,\nu\rangle \models M \,\} \\
\boldsymbol{\gamma}(\, \langle M, P\rangle\,) &= \{\, (\sigma,\nu) \mid (\sigma,\nu) \in \boldsymbol{\gamma}(M) \;\wedge\; \nu \in \boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P) \,\} \\
\boldsymbol{\gamma}(\, \langle E, M, P\rangle\,) &= \{\, (\theta,\sigma) \mid \exists\nu,\ \theta = \nu \circ E \;\wedge\; (\sigma,\nu) \in \boldsymbol{\gamma}(\langle M, P\rangle) \,\}
\end{aligned}$$

I write $\boldsymbol{\gamma}_{\mathbb{P}^\sharp}$ for the concretization function in the base domain, which yields a set of satisfying valuations (independent of a store).

3.3.3 Invariant Checking Code as Inductive Predicates

With the meaning of the memory abstraction defined, we can make a more precise connection between the validation expressions of the checker definition language (Figure 3.3) and the abstract memory states they describe (Figure 3.6). Figure 3.8 defines a syntactic

$e \Rsh \langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle$	e compiles to $\langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle$.
$s \Rsh \langle M, F\rangle$	s compiles to $\langle M, F\rangle$.
$t \Rsh F$	t compiles to F.

$$\frac{e \Rsh \langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle}{\textsf{let } \beta = \alpha\,.\,f \textsf{ in } e \Rsh \langle \alpha@f \mapsto \beta * M_0, F_0\rangle \vee \cdots \vee \langle \alpha@f \mapsto \beta * M_n, F_n\rangle}$$

$$\frac{t \Rsh F \quad e_0 \Rsh \bigvee_{i\in 0..n} \langle M_i, F_i\rangle \quad e_1 \Rsh \bigvee_{j \in 0..m} \langle M_j, F_j\rangle}{\textsf{if } t \textsf{ then } e_0 \textsf{ else } e_1 \Rsh \left(\bigvee_{i\in 0..n} \langle M_i, F \wedge F_i\rangle\right) \vee \left(\bigvee_{j\in 0..m} \langle M_j, \neg F \wedge F_j\rangle\right)}$$

$$\frac{}{\alpha\,.\,c(\delta) \Rsh \langle \alpha.c(\delta), \textsf{true}\rangle} \qquad \frac{s_0 \Rsh \langle M_0, F_0\rangle \quad s_1 \Rsh \langle M_1, F_1\rangle}{s_0 \textsf{ and } s_1 \Rsh \langle M_0 * M_1, F_0 \wedge F_1\rangle}$$

$$\frac{t \Rsh F_0 \quad s \Rsh \langle M, F_1\rangle}{\textsf{let } \alpha = t \textsf{ in } s \Rsh \langle M, (\alpha = F_0) \wedge F_1\rangle} \qquad \frac{t \Rsh F}{t \Rsh \langle \textsf{emp}, F\rangle}$$

$$\frac{}{u \Rsh u} \qquad \frac{}{\alpha \Rsh \alpha} \qquad \frac{t_0 \Rsh F_0 \quad t_1 \Rsh F_1}{t_0 = t_1 \Rsh F_0 = F_1}$$

$$\begin{array}{lll}
\Rsh(e) \stackrel{\text{def}}{=} \langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle & \text{if} & e \Rsh \langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle \\
\Rsh(s) \stackrel{\text{def}}{=} \langle M, F\rangle & \text{if} & s \Rsh \langle M, F\rangle \\
\Rsh(t) \stackrel{\text{def}}{=} F & \text{if} & t \Rsh F \\
\qquad\qquad \Rsh(\pi\,.\,c(\rho) \;:=\; e) & \stackrel{\text{def}}{=} & \pi.c(\rho) := \Rsh(e)
\end{array}$$

Figure 3.8: Translation from validation expressions to formulas.

translation from the normalized checker definition language (Figure 3.3(b)) to the formula form used in this section. We see that the translation $\Lsh(e)$ is defined on all expressions. The normalization process has made this translation particularly direct. As alluded to earlier, we can see that the unfolded region (i.e., the series of points-to edges) correspond to finite access paths from the parameters. Additionally, the existentials are only for the values along those access paths, as they come from the **let**-bindings.

Example 3.6 (A two-level skip list checker formula)**.** Here, we have the formula version of the two-level skip list checker from Figure 2.4(b) and Example 3.4.

$$
\begin{aligned}
\pi.\mathsf{skip1}() \quad &:= \quad \langle \mathsf{emp}, \pi = \mathsf{null} \rangle \\
&\quad\;\; \vee \langle \pi@\mathsf{skip} \mapsto \beta * \pi@\mathsf{next} \mapsto \gamma * \beta.\mathsf{skip1}() * \gamma.\mathsf{skip0}(\beta), \pi \neq \mathsf{null} \rangle \\
\pi.\mathsf{skip0}(\rho) \quad &:= \quad \langle \mathsf{emp}, \pi = \rho \rangle \\
&\quad\;\; \vee \langle \pi@\mathsf{skip} \mapsto \beta * \pi@\mathsf{next} \mapsto \gamma * \gamma.\mathsf{skip0}(\rho), \pi \neq \mathsf{null} \wedge \beta = \mathsf{null} \rangle
\end{aligned}
$$

Example 3.7 (A doubly-linked list checker formula)**.** Below is the formula version of the doubly-linked list checker from Example 3.1 and Example 3.5.

$$
\begin{aligned}
\pi.\mathsf{dll}(\rho) \quad &:= \quad \langle \mathsf{emp}, \pi = \mathsf{null} \rangle \\
&\quad\;\; \vee \langle \pi@\mathsf{prev} \mapsto \beta * \pi@\mathsf{next} \mapsto \gamma * \gamma.\mathsf{dll}(\pi), \pi \neq \mathsf{null} \wedge \beta = \rho \rangle
\end{aligned}
$$

As noted before, we view a checker as summarizing the memory regions where an evaluation of it on that region would succeed. That is, a checker abstracts the set of concrete stores where its evaluation on the store consumes all cells and yields the **true** value. To capture this intuition, we want to show the following:

If $\nu \vdash \langle \sigma, e \rangle \Downarrow \langle [\cdot], \mathbf{true} \rangle$*, then* $\langle \sigma, \nu' \rangle \models \Lsh(e)$ *(for some* $\nu' \supseteq \nu$*).*

In other words, if the validation expression e succeeds and consumes the store, then the store is satisfied by the formula translation (with a possible extension in the valuation).

To prove this statement, we need to discuss the relation between the input store and the residual store. Intuitively, if evaluation of an expression succeeds beginning with a store σ and leaves a residual σ', then the evaluation describes the part of σ where σ' is removed (i.e., describes σ'' where $\sigma = \sigma'' * \sigma'$).

As in Figure 3.7, I leave the understanding of ν satisfying a first-order formula F informal. To relate validation expressions with abstract memory states, we assume that the evaluation and translation of terms t correspond with satisfaction. That is, for whatever terms we consider, evaluation and translation should be defined so that the following hold:

1. *If* $\nu \vdash t \Downarrow \mathsf{true}$*, then* ν *satisfies* $\Rsh(t)$*.*
2. *If* $\nu \vdash t \Downarrow \mathsf{false}$*, then* ν *satisfies* $\neg(\Rsh(t))$*.*
3. *If* $\nu \vdash t \Downarrow u$*, then* $\nu, \alpha \to u$ *satisfies* $\alpha = \Rsh(t)$*.*

Also, Figure 3.7 does not directly define $\models$ for $\langle M, F\rangle$ and $\langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle$. They can be defined as follows:

$$\begin{array}{lll}
\langle \sigma, \nu\rangle \models \langle M, F\rangle & \text{iff} & \langle \sigma, \nu\rangle \models M \text{ and } \nu \text{ satisfies } F \\
\langle \sigma, \nu\rangle \models \langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle & \text{iff} & \text{there is a } \langle M_i, F_i\rangle \ (i \in 0..n) \text{ such that} \\
 & & \langle \sigma, \nu\rangle \models \langle M_i, F_i\rangle
\end{array}$$

Now, we can state the correspondence between the evaluation of validation expressions with the states they describe.

Theorem 3.3 (Successful evaluations correspond to abstract memory states)**.**

> *If* $\nu \vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', \mathsf{true}\rangle$*, then* $\sigma = \sigma'' * \sigma'$ *and* $\langle \sigma'', \nu'\rangle \models \Rsh(e)$ *(for some* $\nu' \supseteq \nu$ *and some* σ''*).*

In particular, if $\nu \vdash \langle \sigma, e\rangle \Downarrow \langle [\cdot], \mathsf{true}\rangle$*, then* $\langle \sigma, \nu'\rangle \models \Rsh(e)$ *(for some* $\nu' \supseteq \nu$*).*

Proof. By induction on the given derivation. See Theorem A.1. $\square$

3.3.4 Properties of Inductive Segments

From the intuitive description of segments in Section 3.3.1, it might be tempting to consider segments as a case of separating implication:

$$\alpha.c(\delta) \mathrel{-\!\!*} \alpha'.c'(\delta') \ ,$$

which describes a region where conjoining any disjoint region that satisfies $\alpha'.c'(\delta')$ (i.e., the remainder) makes that region satisfy $\alpha.c(\delta)$ (i.e., the complete structure) using the standard meaning of $\mathrel{-\!\!*}$ from separation logic[3]. While sufficient and useful for summarizing, this definition of segments is not tight enough to allow for unfolding. In particular, $\alpha.c(\delta) \mathrel{-\!\!*} \alpha'.c'(\delta')$ does not guarantee that $\alpha'.c'(\delta')$ is reachable from $\alpha.c(\delta)$ through checker unfoldings (i.e., $\alpha'.c'(\delta')$ is some "recursive call"). The simplest example is to consider two checkers chk and chk' with different names but the same structure definition (e.g., two separate definitions for a singly-linked list over the same next field): $\alpha.\mathsf{chk}() \mathrel{-\!\!*} \alpha'.\mathsf{chk}'()$ describes some concrete stores, whereas $\alpha.\mathsf{chk}() \mathrel{*\!\!=} \alpha'.\mathsf{chk}'()$ describes no stores. Even if we rule out this case syntactically, there are similar situations that can be constructed with checkers that use the same fields or with the additional parameters.

While inductive segments $\mathrel{*\!\!=}$ are specialized to inductive checkers, it still has some of the properties of the separating implication $\mathrel{-\!\!*}$ (when applied to checkers). We want to be able to unfold segments, but we also need to maintain the properties useful and necessary for analysis. The restrictions in the definition of the 0-step segment $\mathrel{*\!\!=}^0$ (3.1b) are critical here. In particular, we can prove that $\mathrel{*\!\!=}$ is a restriction of $\mathrel{-\!\!*}$.

Lemma 3.4 (Stronger than separating implication)**.**

[3]More formally, $\langle \sigma * \sigma', \nu \rangle \models M \mathrel{-\!\!*} M'$ iff for all σ' (disjoint from σ), if $\langle \sigma', \nu \rangle \models M'$, then $\langle \sigma, \nu \rangle \models M$.

If $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \mathrel{\ast\!\!=} \alpha'.c'(\delta')\,)$, *then* $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \mathrel{\ast\!\!-} \alpha'.c'(\delta')\,)$.

Proof. By induction on the length of the segment. □

Corollary 3.5 (Elimination of segments)**.** *As a consequence, we get an elimination rule.*

If $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \mathrel{\ast\!\!=} \alpha'.c'(\delta') \ast \alpha'.c'(\delta')\,)$, *then* $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta)\,)$.

Proof. Direct by Lemma 3.4. □

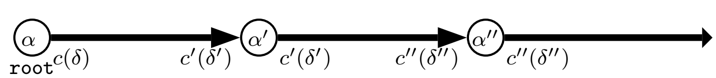
For analysis, it is also necessary to have the following basic but important properties.

Lemma 3.6 (Combining segments)**.**

If $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \mathrel{\ast\!\!=} \alpha'.c'(\delta') \;\ast\; \alpha'.c'(\delta') \mathrel{\ast\!\!=} \alpha''.c''(\delta'')\,)$,
then $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \mathrel{\ast\!\!=} \alpha''.c''(\delta'')\,)$.

Proof. By induction on the length of the first segment. □

In terms of the shape graph, these facts in Corollary 3.5 and Lemma 3.6 allow the analysis to discard intermediate nodes when they are no longer needed, such as α' and α'' in the following graph:



It allows us to discover when a region again satisfies a complete run of a checker (i.e., the entire structure again adheres to the data structure invariant). In the above, dropping the intermediate nodes allows us to derive that the region from α (pointed to by `root`) satisfies the invariants of checker c.

This last fact allows us to introduce segments anywhere when needed.

Lemma 3.7 (Introduction of empty segments)**.**

For all ν, $([\cdot], \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \ast\!\!= \alpha.c(\delta)\,)$.

Proof. Direct by definition (case (3.1b)). □

Note that our notion of segments is designed to capture precisely a partial run (i.e., a partial derivation) and not only structure segments where an empty structural segment need not necessarily have equal additional parameters (i.e., δ's). This distinction shows up in the above properties.

3.4 Typing Checker Parameters

As alluded to in Section 2.5, before we perform a shape analysis on the program code using the user-supplied checker definitions, we first perform a separate type analysis on the checker definitions themselves. This type analysis gathers information that we then use to guide the abstract interpretation over the program code. In particular, we saw that at program point 21 in Figure 3.2, in order to materialize the fields of γ (i.e., the node pointed to by `pa`), we need to unfold the segment between α and β "backward" from β. However, note that this reasoning was based only on our intuitive understanding of the `rbtree` checker. Nevertheless, we notice that with some additional type information on the checker parameters, the analysis can then make this unfolding decision automatically.

In this section, we prescribe a type system to checker parameters that classifies them into various kinds of pointers and non-pointers. This type information will then instruct the shape analysis where to perform unfolding (see Section 4.1.2). Intuitively, we consider a checker parameter to have pointer type if one of its fields is ever dereferenced along some path in a checker run (on a satisfying store). Thus far in the paper, we have in essence implicitly assigned a pointer type to the traversal parameter (i.e., π in a definition $\pi.c(\rho)$).

To direct unfolding decisions, we introduce further refinements of the pointer type to indicate, for example, which fields f are dereferenced. We thus write a pointer type as a record of fields that may be dereferenced $\{f_0\langle\ell_0\rangle, \ldots, f_n\langle\ell_n\rangle\}$. We consider the non-pointer type to be simply the empty record of fields $\{\}$. The *level* ℓ provides a relative measure of where in the sequence of checker calls a field is dereferenced (i.e., where the points-to edge is materialized). To describe this notion, consider a three-element singly-linked list and the sequence of checker calls that describes it (using the `list` checker from Example 2.1), shown below diagrammatically:

$$\alpha_0 \xrightarrow{\textsf{next}} \alpha_1 \xrightarrow{\textsf{next}} \alpha_2 \xrightarrow{\textsf{next}} \textsf{null} \qquad \alpha_0.\texttt{list}() \cdots \alpha_1.\texttt{list}() \cdots \alpha_2.\texttt{list}() \cdots \textsf{null}.\texttt{list}()$$

The right represents the computation tree of $\alpha_0.\texttt{list}()$ (i.e., the derivation of the inductive predicate). For a pointer argument of a call, such as α_0, we want to track an approximation of where along the run are the fields of α_0 dereferenced (i.e., materialized). For example, if we number the levels of the computation tree as indicated with the subscripts, then we can say that the $\textsf{next}$ field of α_0 is dereferenced in level 0. Intuitively, a level ℓ is thus an integer approximating in how many checker calls is the field dereferenced or $\textsf{unk}$ if unknown (e.g., at different levels depending on some condition).

Example 3.8 (A singly-linked list checker definition with types)**.** The following assigns parameters types to a normalized version of the `list` checker from Example 2.1.

```
(l : {next<0>}).list()  :=
  if (l = null) then true
  else
    let n : {next<1>} = l.next in
    n.list()
```

As shown above, we consider a relative notion of levels where there "current" level is 0; negative integers indicate backward from the current call, while positive integers indicate forward.

Example 3.9 (A doubly-linked list checker definition with types)**.** In the following, we consider parameters types for the `dll` checker from Example 3.5.

```
(l : {prev<0>,next<0>}).dll(l_p : {prev<-1>,next<-1>})  :=
  if (l = null) then true
  else
    let p : {} = l.prev in
    let n : {prev<1>,next<1>} = l.next in
    p = l_p and n.dll(l)
```

In Figure 3.9, I present a type-checking algorithm for the pointer types described above, which will then lead to an inference algorithm. I write Γ for a type environment mapping symbolic values α to types τ. Both the types and levels form (semi-)lattices where the ordering is given by the $<:$ and $<::$ relations, respectively. For types, records are ordered by subset containment of fields modulo ordering in the levels, while the level lattice is the flat lattice on integers with `unk` as the top element. Intuitively, the absence of a field indicates it is not dereferenced anywhere. The core judgment is $\Gamma \vdash e$ ok, which checks that reads in a validation expression are good with respect to the parameter types Γ. For

$$
\begin{array}{lrcl}
\text{types} & \tau & ::= & \{f_0\langle\ell_0\rangle, \ldots, f_n\langle\ell_n\rangle\} \\
\text{levels} & \ell & ::= & k \mid \mathsf{unk}
\end{array}
$$

$\boxed{\Gamma \vdash e\ \mathsf{ok} \qquad \text{Under } \Gamma\text{, field reads in } e \text{ are ok.}}$

$$\frac{\{f\langle 0\rangle\} <: \Gamma(\alpha) \quad \Gamma, \beta : \tau \vdash e\ \mathsf{ok}}{\Gamma \vdash \mathsf{let}\ \beta\ \mathtt{:}\ \tau\ \mathtt{=}\ \alpha\mathtt{.}f\ \mathsf{in}\ e\ \mathsf{ok}}\ \textsf{t-read}$$

$$\frac{\Gamma(\alpha) - 1 = \tau_0 \quad \Gamma(\delta) - 1 = \tau_1 \quad (\mathtt{(}\pi\ \mathtt{:}\ \tau_0\mathtt{)}\mathtt{.}c\mathtt{(}\rho\ \mathtt{:}\ \tau_1\mathtt{)}\ \mathtt{:=}\ e)}{\Gamma \vdash \alpha\mathtt{.}c\mathtt{(}\delta\mathtt{)}\ \mathsf{ok}}\ \textsf{t-call}$$

$$\frac{\Gamma, \alpha : \tau \vdash s\ \mathsf{ok}}{\Gamma \vdash \mathsf{let}\ \alpha\ \mathtt{:}\ \tau\ \mathtt{=}\ t\ \mathsf{in}\ s\ \mathsf{ok}}\ \textsf{t-let-term} \qquad \frac{\Gamma \vdash e_0\ \mathsf{ok} \quad \Gamma \vdash e_1\ \mathsf{ok}}{\Gamma \vdash \mathsf{if}\ t\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1\ \mathsf{ok}}\ \textsf{t-if}$$

$$\frac{\Gamma \vdash s_0\ \mathsf{ok} \quad \Gamma \vdash s_1\ \mathsf{ok}}{\Gamma \vdash s_0\ \mathsf{and}\ s_1\ \mathsf{ok}}\ \textsf{t-and} \qquad \frac{}{\Gamma \vdash t\ \mathsf{ok}}\ \textsf{t-term}$$

$\boxed{\vdash \mathit{chkdef}\ \mathsf{ok} \qquad \text{Field reads in } \mathit{chkdef} \text{ are ok.}}$

$$\frac{\pi : \tau_0, \rho : \tau_1 \vdash e\ \mathsf{ok}}{\vdash \mathtt{(}\pi\ \mathtt{:}\ \tau_0\mathtt{)}\mathtt{.}c\mathtt{(}\rho\ \mathtt{:}\ \tau_1\mathtt{)}\ \mathtt{:=}\ e\ \mathsf{ok}}\ \textsf{t-chkdef}$$

$\boxed{\begin{array}{l} \tau_0 <: \tau_1 \\ \ell_0 <:: \ell_1 \end{array}}$

$$\frac{\ell_i <:: \ell'_i \quad (0 \le i \le n \le m)}{\{f_0\langle\ell_0\rangle, \ldots, f_n\langle\ell_n\rangle\} <: \{f_0\langle\ell'_0\rangle, \ldots, f_m\langle\ell'_m\rangle\}} \qquad \frac{}{\ell <:: \mathsf{unk}} \qquad \frac{}{\ell <:: \ell}$$

Figure 3.9: Type-checking checker parameters.

a field read, we check that the field is in the set of dereferenced fields (rule t-read). For a checker application, we check that the set of dereferenced fields for the actuals and formals are the same, but we need to shift the frame of reference of the levels (rule t-call). I write $\tau - k$ for the function on types that decrements each of the field levels by k (and where unk maps to unk). Finally, at the top-level, we type-check each checker definition separately and for each definition, we check that the body is good.

Inference. The type-checking algorithm is fairly straightforward, but it also yields a natural extension to inferring parameter types by a least fixed-point computation. The inference proceeds by initializing all type declarations to $\{\}$ (the bottom element). At each place where we check conformance in Figure 3.9, we instead compute the join in the type lattice and update the appropriate declarations. Then, we iterate until we reach a fixed point. Since the type lattice has finite height (as the number of fields is fixed for any set of checker definitions), the process terminates. The type assignments shown in the examples are the ones computed by this inference algorithm.

3.4.1 Guidance for Unfolding

This procedure provides fairly generic support for unfolding (forward and backward). The graph unfolding described in Section 4.1.1 combined with this type analysis allows support for back pointers that go back a finite number of steps (e.g., a list with pointers that go back two nodes, a binary tree with parent pointers), but more importantly, it makes the unfolding process less sensitive to how the checkers are written. For example, the alternative doubly-linked list checker of Example 3.10 works equally well.

Example 3.10 (An alternative doubly-linked list checker definition with types). The following checkers define a doubly-linked list where the unfolding materializes the `next` field of the current node and then the `prev` field of the next node (if it is non-null) instead of the `next` and `prev` fields of the current node as in `dll` (Example 3.9).

```
(l : {prev<0>,next<1>}).npdll()  :=
  if (l = null) then true
  else
    let p : {} = l.prev in
    p = null and l.nenpdll()

(l : {prev<-1>,next<0>}).nenpdll()  :=
  let n : {prev<0>,next<1>} = l.next in
  if (n = null) then true
  else
    let np : {} = n.prev in
    np = l and n.nenpdll()
```

The types are slightly different, which in turn guides the graph unfolding algorithm appropriately. In particular, the `prev` field of `l` of `nenpdll` (non-empty next-prev doubly-linked list) is at level `-1` corresponding to our intuitive understanding of this checker that the `prev` field of `l` should have been materialized in the previous call. Also, observe that in t-call, we make no distinction between the traversal parameters and the additional parameters. Thus, with this type information, one could consider checkers with, for example, multiple forward parameters (at least in terms of unfolding).

Note that unlike traditional uses, this type system is not meant to ensure safe evaluation. Rather, it is designed to compute a simple approximation of the possible stores described by a validation expression e that provides guidance for unfolding. It does not, for example, ensure that an evaluation will succeed.

We discuss more precisely the approximation that is computed by this type analysis later in this section, but as it is an approximation, we first consider an example of its limitations. As stated above, this type analysis provides guidance to unfolding for back pointers that go back a fixed number of steps, but it provides little useful information for back pointers that can go arbitrarily far back. For instance, consider an n-ary tree represented as nodes with pointers to its first child, its immediate sibling, and its parent.

Example 3.11 (A child-sibling tree checker definition with types)**.**

```
(t : {parent<unk>,child<unk>,sibling<unk>})
  .cstree(t_p : {parent<unk>,child<unk>,sibling<unk>})  :=
    if (t = null) then true
    else
      let p : {} = l.parent in
      let c : {parent<unk>,child<unk>,sibling<unk>} = l.child in
      let s : {parent<unk>,child<unk>,sibling<unk>} = l.sibling in
      p = t_p and c.cstree(t) and s.cstree(t_p)
```

Note that the issue comes from the `parent` pointer. Without the `parent` pointer, the shape is simply a binary tree. Furthermore, if it instead pointed back to the previous sibling (and the parent in case of the first sibling), then the shape is just like a binary tree with parent pointers. However, in this case, the type inference yields the information that pointer parameters `t` and $\mathtt{t}_\mathrm{p}$ have `parent`, `child`, and `sibling` fields but at unknown levels. Our analysis can still use this checker to analyze code that walks down the tree (i.e., forward in the checker traversal) to perform an operation, but it does not have the guidance to analyze code that starts from an arbitrary node in the tree and walks up the tree along `parent` pointers. A possibility to support such structures may be to strengthen the type analysis by refining the lattice of levels and utilize that additional information in unfolding.

3.4.2 Approximation of Checker Evaluations

The goal with this type analysis is to compute for each checker parameter an approximation of its fields and where along a successful evaluation of a validation expression are those fields read. Where the fields are read correspond to where the analysis should look to materialize them. Intuitively, in a successful evaluation, a field is read and consumed at particular time (i.e., level) in the execution and through a specific pointer variable (as alluded to in Section 3.2.2 and Section 3.3.3). A type assignment to a parameter $\pi : \tau$ should be viewed as saying π may read from the fields in τ at the specified levels (relative to the current level). That is, the type τ can always be extended with additional possible fields. There is also an approximation due to aliasing, as there could be arbitrary aliasing in the store or amongst the parameters. In particular, the inference algorithm says that a pointer π has a field f only if it witnesses a read of f through π.

To capture the notion of levels, we can instrument the evaluation semantics from Figure 3.5 to essentially time-stamp when a field is read. I write $\bar{\sigma}$ for a time-stamped store where each cell

$$[u_0 \mapsto u_1]_{stamp}$$

is marked with a time stamp that can be either any to indicate an "unconsumed" cell or $\alpha{\cdot}k$ for a cell read through α at level k. In Figure 3.10, I show the instrumented evaluation semantics where each step is marked with the current level. The rules of interest are $\mathsf{ie\text{-}read}$

time stamps $\quad stamp \ ::= \ \mathsf{any} \mid \alpha{\cdot}k$

$\nu \vdash \langle \bar{\sigma}, e \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle$	Under valuation ν and in time-stamped store $\bar{\sigma}$ at k, expression e leaves a residual store $\bar{\sigma}'$ and evaluates to value u.

$$\frac{\nu, \beta \to u \vdash \langle [\nu(\alpha) + f \mapsto u]_{\alpha\cdot k} * \bar{\sigma}, e \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}{\nu \vdash \langle [\nu(\alpha) + f \mapsto u]_{\mathsf{any}} * \bar{\sigma}, \mathsf{let}\ \beta = \alpha\,.\,f\ \mathsf{in}\ e \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}\ \textsf{ie-read}$$

$$\frac{\nu \vdash t \Downarrow \mathsf{true} \quad \nu \vdash \langle \bar{\sigma}, e_0 \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}{\nu \vdash \langle \bar{\sigma}, \mathsf{if}\ t\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1 \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}\ \textsf{ie-if-t} \qquad \frac{\nu \vdash t \Downarrow \mathsf{false} \quad \nu \vdash \langle \bar{\sigma}, e_1 \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}{\nu \vdash \langle \bar{\sigma}, \mathsf{if}\ t\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1 \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}\ \textsf{ie-if-f}$$

$$\frac{\nu \vdash \langle \bar{\sigma}, [\alpha, \delta/\pi, \rho]e \rangle_{k+1} \Downarrow \langle \bar{\sigma}', u \rangle \quad (\pi\,.\,c(\rho)\ := e)}{\nu \vdash \langle \bar{\sigma}, \alpha\,.\,c(\delta) \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}\ \textsf{ie-call}$$

$$\frac{\nu \vdash \langle \bar{\sigma}, s_0 \rangle_k \Downarrow \langle \bar{\sigma}', u_0 \rangle \quad \nu \vdash \langle \bar{\sigma}', s_1 \rangle_k \Downarrow \langle \bar{\sigma}'', u_1 \rangle}{\nu \vdash \langle \bar{\sigma}, s_0\ \mathsf{and}\ s_1 \rangle_k \Downarrow \langle \bar{\sigma}'', u_0 \wedge u_1 \rangle}\ \textsf{ie-and}$$

$$\frac{\nu \vdash t \Downarrow u_0 \quad \nu, \alpha \to u_0 \vdash \langle \bar{\sigma}, s \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}{\nu \vdash \langle \bar{\sigma}, \mathsf{let}\ \alpha = t\ \mathsf{in}\ s \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle}\ \textsf{ie-let-term} \qquad \frac{\nu \vdash t \Downarrow u}{\nu \vdash \langle \bar{\sigma}, t \rangle_k \Downarrow \langle \bar{\sigma}, u \rangle}\ \textsf{ie-term}$$

Figure 3.10: Instrumented evaluation of validation expressions with time-stamped reads.

$\vdash \langle \bar{\sigma}, \nu \rangle : \Gamma$	State $\langle \bar{\sigma}, \nu \rangle$ is approximated by Γ.

$$\frac{\mathrm{dom}(\nu) = \mathrm{dom}(\Gamma)}{\vdash \langle [\cdot], \nu \rangle : \Gamma}\ \textsf{st-emp}$$

$$\frac{\{f\langle k \rangle\} <: \Gamma(\alpha) \quad \vdash \langle \bar{\sigma}, \nu \rangle : \Gamma}{\vdash \langle [\nu(\alpha) + f \mapsto u]_{\alpha\cdot k} * \bar{\sigma}, \nu \rangle : \Gamma}\ \textsf{st-stamped} \qquad \frac{\vdash \langle \bar{\sigma}, \nu \rangle : \Gamma}{\vdash \langle [u_0 \mapsto u_1]_{\mathsf{any}} * \bar{\sigma}, \nu \rangle : \Gamma}\ \textsf{st-any}$$

Figure 3.11: The relationship between time-stamped stores and the type environments that approximate them.

and $\mathsf{ie\text{-}call}$. For a field read, the input store must have a cell for that field that has not yet been stamped, which is then time-stamped for further evaluation and thus cannot be read again. On a checker call, we simply increment the current level. The remaining rules are as before in Figure 3.5 except the current level is carried around; I show them here only for completeness. Intuitively, evaluation starts with a store $\bar{\sigma}$ of cells and ends with a store $\bar{\sigma}'$ that consists of the same cells but are now stamped with the level when and the pointer through which they were read.

The correspondence between evaluation and instrumented evaluation is rather straightforward. To state it, we simply define conversions between stores and time-stamped stores that either remove time-stamped cells or mark all cells as any, respectively:

$$
\begin{array}{cc}
|\,[\cdot]\,| = [\cdot] & |\,[\cdot]\,|_{\mathsf{any}} = [\cdot] \\
|\,[u_0 \mapsto u_1]_{\mathsf{any}} * \bar{\sigma}\,| = [u_0 \mapsto u_1] * |\,\bar{\sigma}\,| & |\,[u_0 \mapsto u_1] * \sigma\,|_{\mathsf{any}} = [u_0 \mapsto u_1]_{\mathsf{any}} * |\,\sigma\,|_{\mathsf{any}} \\
|\,[u_0 \mapsto u_1]_{\alpha\cdot k} * \bar{\sigma}\,| = |\,\bar{\sigma}\,| &
\end{array}
$$

We can now state that instrumented evaluation is sound and complete with respect to evaluation.

Theorem 3.8 (Correspondence between evaluation and instrumented evaluation)**.**

1. *If* $\nu \vdash \langle \bar{\sigma}, e\rangle_k \Downarrow \langle \bar{\sigma}', u\rangle$*, then* $\nu \vdash \langle |\,\bar{\sigma}\,|, e\rangle \Downarrow \langle |\,\bar{\sigma}'\,|, u\rangle$*.*
2. *If* $\nu \vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', u\rangle$*, then* $\nu \vdash \langle \bar{\sigma}_r * |\,\sigma\,|_{\mathsf{any}}, e\rangle_k \Downarrow \langle \bar{\sigma}'_r * |\,\sigma'\,|_{\mathsf{any}}, u\rangle$
   (for all $\bar{\sigma}_r, k$ *and for some* $\bar{\sigma}'_r$*).*

Proof. By induction on the given derivations (separately). □

From instrumented evaluation to evaluation (fact 1) simply requires stripping off the instrumentation. The other direction (fact 2) states that we can add back additional disjoint cells that may record history information.

With an evaluation instrumented with a history of field reads, we can describe more precisely the approximation computed by the type analysis. Figure 3.11 states the relationship between time-stamped stores $\bar{\sigma}$ and type environments Γ (by a relation defined by induction on the store). The type environment Γ may indicate more fields than are present in the concrete store (from st-emp), but a cell that has been read and stamped should have a corresponding entry in Γ (from st-stamped). Any cell that has not been read requires no constraint in Γ (from st-any). Aliasing is approximated in a rather coarse manner, as there are no constraints on aliases to time-stamped reads. Specifically, rule st-stamped looks only at α and not any aliases of α (i.e., a β where $\nu(\alpha) = \nu(\beta)$). Though a technical point, it is thus important that the binding of actuals to formals in the evaluation of a checker call (ie-call in Figure 3.10) is defined by substitution. Finally, we can state that a type environment Γ computed on a validation expression e is an approximation of stores described by e relative to the current level in evaluation.

Theorem 3.9 (Typing computes an approximation of time-stamped stores)**.** *Assume that each checker definition has been type-checked.*

$$\textit{If } \nu \vdash \langle \bar{\sigma}, e \rangle_k \Downarrow \langle \bar{\sigma}', u \rangle \;,\; \Gamma \vdash e \;\textsf{ok} \;,\; \textit{and } \vdash \langle \bar{\sigma}, \nu \rangle : \Gamma + k \;,$$
$$\textit{then } \vdash \langle \bar{\sigma}', \nu' \rangle : \Gamma' + k \quad \textit{(for some } \nu' \supseteq \nu \textit{ and } \Gamma' \supseteq \Gamma\textit{).}$$

In the above, $\Gamma + k$ is the function that increments all the levels in Γ by k (i.e., lifting the function on types).

Proof. By induction on the first two derivations and case analysis on last one. See Theorem A.3. ∎

Chapter 4

Analysis Algorithm

Our analysis works as a typical abstract interpretation on programs. In particular, the analyzer computes an abstract state

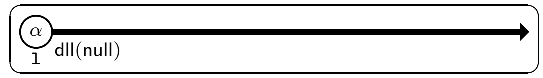
$$A \quad = \quad \langle E_0, M_0, P_0 \rangle \vee \cdots \vee \langle E_n, M_n, P_n \rangle$$

for each control point. To do that, we need abstract transformers for commands, such as assignment and condition testing. The key domain operation is the unfolding of checker edges (forward and backward) in order to *materialize* points-to edges. A novel aspect of our algorithm is that we reduce the problem of unfolding segments backward to unfolding them forward (Section 4.1). To infer loop invariants and obtain a terminating analysis, we define comparison and join operations on abstract states that summarize graphs by folding them into checker edges. The primary source of complexity in folding is the interaction between the shape graph and the base data domain. A nice property of our algorithm is that at a high-level, we separate the computation of the comparison and join operations into phases: first, a traversal over the shape graph gathering constraints, and then, a propagation of these constraints to the base data domain before applying the corresponding base domain operation (Section 4.2). Like others, we materialize as needed to reflect updates and dereferences, but instead of summarizing eagerly, we delay folding in order to use history information to guide the process.

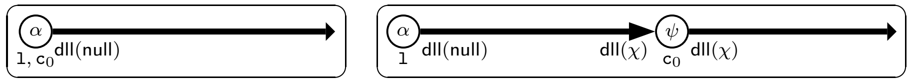
4.1 Abstract Transition with Unfolding

Recall that the complete checker edges and partial checker edges summarize memory regions. In order to reflect memory updates in a precise manner, we often need to partially concretize these summaries, which we do by unfolding. To describe unfolding in detail, we consider a schematic example of doubly-linked list traversals that illustrate the various forms of unfolding (shown in Figure 4.1). Initially, we have that `l` points to the head of a doubly-linked list (satisfying the dll checker from Example 3.1). From program point 3 to point 4, we perform a forward unfolding of a complete checker edge to materialize the edge corresponding to $\texttt{c}_0\texttt{->next}$ (the one from α in the first iteration and the one from ψ in the fixed-point iteration). Observe that the analysis determines that it should, for example, unfold forward at α in the first iteration because while there is no outgoing points-to edge for the next field (corresponding to $\texttt{c}_0\texttt{->next}$), node α does have an outgoing checker edge (i.e., α is the traversal argument to a checker). Because checkers are inductive definitions where points-to edges emanate from the traversal parameter, unfolding the inductive predicate at its traversal argument (e.g., α in $\alpha.\mathsf{dll}(\mathsf{null})$) is likely to materialize the desired points-to edge.

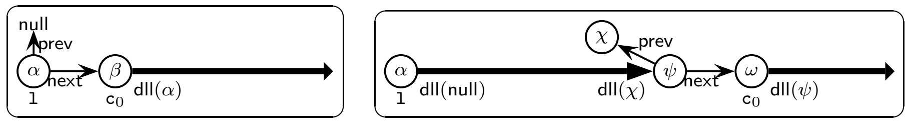
From program point 7 to point 8, to materialize the edge $\texttt{c}_1\texttt{->next}$, we must unfold



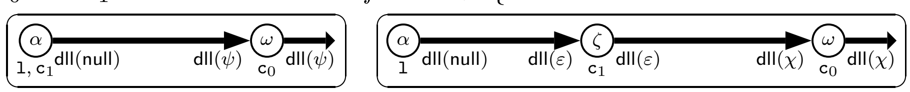
```
c0 = l;
while (c0 != null && c0 should be advanced forward) {
```



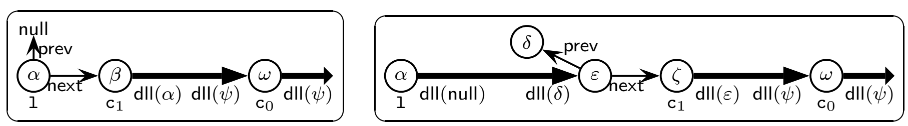
```
  c0 = c0->next;
```



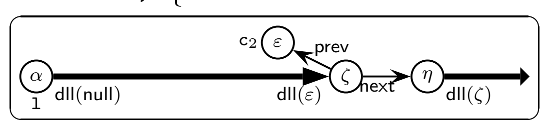
```
}
c1= l;
while (c1 != c0 && c1 should be advanced forward) {
```



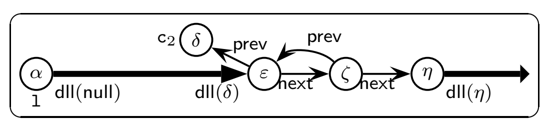
```
  c1 = c1->next;
```



```
}
c2 = c1 != null ?  c1->prev :  null;
while (c2 != null && c2 should be advanced backward) {
```



```
  c2 = c2->prev;
```



```
}
```

first iteration | *at fixed point*

Figure 4.1: Various forms of unfolding for doubly-linked lists (checker dll from Example 3.1). The first loop shows forward unfolding of a complete checker edge, the second loop presents forward unfolding of a partial checker edge, and the third loop demonstrates backward unfolding of a partial checker edge.

forward at the head of a dll segment (the one from α to ω in the first iteration and the one from ζ to ω in the fixed-point iteration). Like in the previous case, the analysis determines it should unfold forward because it arrives at an outgoing checker edge, but in this case, it is a partial checker edge and thus how to unfold it does not come directly from a user-supplied inductive definition.

Finally, from program point 11 to point 12, we can only make progress by unfolding backward at the tail of a segment (the one from α to ζ) to materialize the edge for $\texttt{c}_2\texttt{->prev}$. There are two distinct aspects to both forward and backward unfolding (though they are more evident in the backward case):

1. implementing the unfolding operations on the abstract domain, while justifying their soundness (described in Section 4.1.1) and

2. determining when to apply which unfolding operation. (discussed in Section 4.1.2).

Unlike forward unfolding, determining when and where to apply backward unfolding is not so obvious.

4.1.1 Unfolding Operations

We apply an unfolding in order to expose a heap cell (i.e., a points-to edge) before performing an operation on it. Checker definitions include not only shape information (i.e., how are the points-to edges arranged) but also data information in the form of pure constraints on the exposed heap cells. Thus, unfolding not only modifies the shape graph M but must also add additional constraints to the pure state P. In general, unfolding takes an element of the product domain $\langle M, P\rangle$ and yields a disjunction of abstractions

$\langle M'_0, P'_0\rangle \vee \cdots \vee \langle M'_n, P'_n\rangle$. Though, it is often the case that all but one of the disjuncts are ruled out by the data domain (i.e., we derive a contradiction in the pure constraints).

To mark the phases of unfolding, we distinguish an unfolding operation that works only on shape graphs from the overall unfolding operation. This distinction is also useful in describing other domain operations that rely on unfolding. I write $\mathbf{u}_\alpha$ for the forward unfolding at node α that takes a shape graph M and returns a set of graph and formula pairs where the first-order formulas give the pure constraints on the unfolded graphs. The overall unfolding operation $\mathbf{unfold}_\alpha$ then takes an element of the product domain $\langle M, P\rangle$ and returns a disjunction of such elements.

Unfolding Inductive Checkers

Let us consider the unfolding of a complete checker edge $\alpha.c(\delta)$, in the abstract element $\langle M * \alpha.c(\delta), P\rangle$. I describe unfolding of complete checker edges in detail, as it serves as a basis for the unfolding of segments.

Definition 4.1 (Unfolding of inductive checkers)**.** *The unfolding of a checker proceeds by unfolding each checker rule separately:*

$$\begin{aligned}\mathbf{unfold}_\alpha(\langle M * \alpha.c(\delta), P\rangle) &\stackrel{\text{def}}{=} \bigvee_{R \in c} \mathbf{unfold}_\alpha(\langle M * \alpha.R(\delta), P\rangle)\\ \mathbf{u}_\alpha(M * \alpha.c(\delta)) &\stackrel{\text{def}}{=} \bigcup_{R \in c} \mathbf{u}_\alpha(M * \alpha.R(\delta))\end{aligned}$$

where I write $R \in c$ for a checker rule R of checker definition c and overload checker application to also apply to rules.

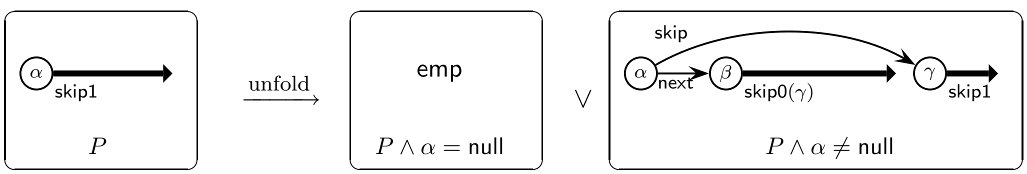
Definition 4.2 (Unfolding of checker rules)**.** *To unfold a checker rule, we first materialize* *the fields of the rule and then add the data constraint.*

$$\mathbf{unfold}_\alpha(\langle M * \alpha.R(\delta), P\rangle) \stackrel{\text{def}}{=} \langle M', \mathbf{guard}_{\mathbb{P}^\sharp}(P, F')\rangle$$
$$\text{where } \{ (M', F') \} = \mathbf{u}_\alpha(M * \alpha.R(\delta))$$
$$\mathbf{u}_\alpha(M * \alpha.R(\delta)) \stackrel{\text{def}}{=} \left\{ (M * [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}](M^{\mathrm{u}} * M^{\mathrm{f}}), [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]F) \right\}$$

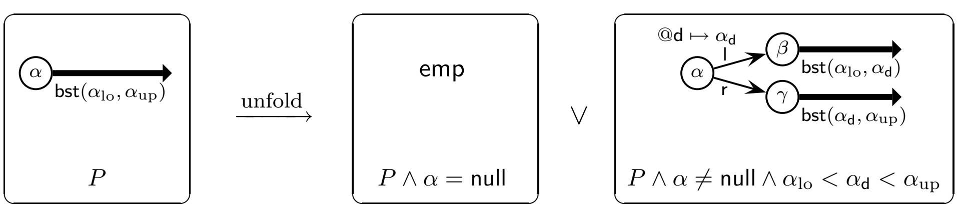
We assume the base domain provides a $\mathbf{guard}_{\mathbb{P}^\sharp}(P, F)$ *function that is a sound approximation of constraining* P *with* F*. In the shape graph, we simply unfold the checker rule and perform substitutions (where* $\vec{\varepsilon}$ *are fresh; and rule* R *has formals* π *and* ρ*, has free variables* $\vec{\kappa}$*, and has the form* $\langle M^{\mathrm{u}} * M^{\mathrm{f}}, F\rangle$*).*

This scheme can be performed in an automatic way and generates a finite number of disjuncts, which are well-formed elements of the domain.

Example 4.1 (Unfolding a skip list)**.** I exhibit an unfolding of the skip1 checker from Figure 2.4. The addition of the pure constraints is shown explicitly.



Example 4.2 (Unfolding a binary search tree)**.** I show the unfolding of a region satisfying the bst checker from Example 3.2.



Forward Unfolding of Inductive Segments

Because the semantics of a partial checker edge

$$\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$$

is defined by induction over the sequence of derivation steps (from α to α'), we can define an unfolding scheme analogous to the one for complete checker edges. We call this operation *forward unfolding* since it proceeds by unfolding checker definitions at the top of the derivation tree in the "standard" way (i.e., corresponding to the materialization of edges at the head α). This unfolding operation is exactly what is needed to materialize the edge for `c`$_1$`->next` at program point 7 in Figure 4.1.

We extend the definition of forward unfolding ($\mathbf{unfold}_\alpha$) for segments. Like for complete checker edges, $\mathbf{unfold}_\alpha$ on partial checker edges generates a finite disjunction of $\langle M, P\rangle$ pairs. However, for partial checker edges, we must consider an additional case for the empty segment (i.e., the 0-step segment); only if the segment is non-empty (i.e., is of 1-or-more steps) do we get materializations corresponding to the checker rules of c.

Definition 4.3 (Forward unfolding of segments)**.**

$$
\begin{aligned}
&\mathbf{unfold}_\alpha(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \\
&\quad\stackrel{\text{def}}{=}\ \mathbf{unfold}^0_\alpha(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \vee \\
&\qquad\qquad\qquad \left(\bigvee_{R \in c} \mathbf{unfold}_\alpha(\langle M * \alpha.R(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \right)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{unfold}^0_\alpha(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \\
&\quad\stackrel{\text{def}}{=}\ \begin{cases} \langle M, \mathbf{guard}_{\mathbb{P}^\sharp}(P, \alpha = \alpha' \wedge \delta = \delta')\rangle & \textit{if} \quad c = c' \\ \bot & \textit{if} \quad c \neq c' \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{u}_\alpha(M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')) \\
&\quad\stackrel{\text{def}}{=}\ \mathbf{u}^0_\alpha(M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')) \cup \left(\bigcup_{R \in c} \mathbf{u}_\alpha(M * \alpha.R(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')) \right)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{u}^0_\alpha(M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')) \\
&\quad\stackrel{\text{def}}{=}\ \begin{cases} \{\, (M, \alpha = \alpha' \wedge \delta = \delta')\,\} & \textit{if} \quad c = c' \\ \{\} & \textit{if} \quad c \neq c' \end{cases}
\end{aligned}
$$

Observe that for the empty segment ($\mathbf{unfold}^0_\alpha$), we assert the additional equalities ($\alpha = \alpha'$ and $\delta = \delta'$) in the base domain. Only with these equalities can we determine in the analysis that the segment at program point 7 of the example in Figure 4.1 is non-empty. To maintain a fully reduced representation in the shape graph, we can merge the nodes of an empty segment (i.e., α with α' and δ with δ').

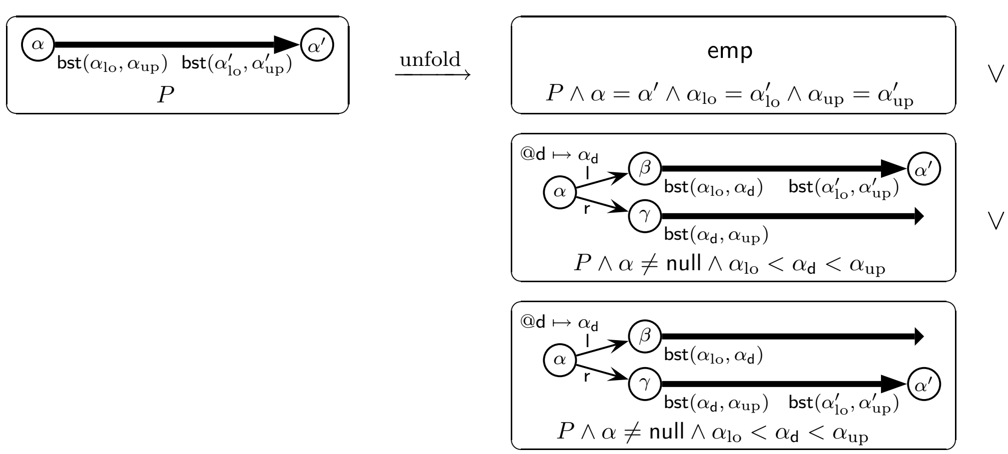
The definition for $\mathbf{unfold}_\alpha$ on checker rules for segments are like for complete checker edges. They follow directly from the semantics given in Figure 3.7.

Definition 4.4 (Forward unfolding of checker rules for segments)**.**

$$\begin{aligned}
&\mathbf{unfold}_\alpha(\langle M * \alpha.R(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \\
&\quad\stackrel{\text{def}}{=} \bigvee_{(M',F') \in \mathbf{u}_\alpha(M * \alpha.R(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'))} \langle M', \mathbf{guard}_{\mathbb{P}^\sharp}(P, F')\rangle \\
&\mathbf{u}_\alpha(M * \alpha.R(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')) \\
&\quad\stackrel{\text{def}}{=} \big\{\, (M * [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}](M^{\mathrm{u}} * M^{\mathrm{f}} * \beta.c''(\gamma) \mathrel{*\!\!=} \alpha'.c'(\delta')), [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]F) \;\big|\; \\
&\qquad\qquad R = \langle M^{\mathrm{u}} * (M^{\mathrm{f}} * \beta.c''(\gamma)), F\rangle \textit{ with formals } \pi \textit{ and } \rho\textit{; and } \vec{\varepsilon} \textit{ are fresh} \big\}
\end{aligned}$$

Note that in the shape unfolding ($\mathbf{u}_\alpha$), there is a case for each recursive checker call $\beta.c''(\gamma)$ (i.e., the segment could end in any of the branches).

Example 4.3 (Forward unfolding a binary search tree segment)**.** I show the unfolding of a segment region satisfying the bst checker from Example 3.2.



Note that using separating implication ($-\!\!*$) for partial checker edges would make them very difficult to unfold, as it would require involved restrictions to be made on checkers. Instead, our notion of segments as we define here ($*\!\!=$) seems to be closer to our intuitive understanding of partial derivations of checkers and thus leads to a natural forward unfolding operation.

Backward Unfolding of Inductive Segments

The unfolding function defined above allows the analysis to materialize memory regions from the traversal argument of a checker edge. However, these unfolding operations do not apply to algorithms walking backward through invertible data structures, such as doubly-linked lists, as the sequence of edge dereferences does *not* follow the recursive checker calls that the forward unfolding would uncover. For example, this situation arises at program point 11 in Figure 4.1. From our intuitive understanding of dll, we know that if the segment between α and ζ is non-empty, then ε—the value of $\mathtt{c}_2$—lies along that segment "just before ζ" (i.e., ε's next field points to ζ). Thus, if we are able to *unfold backward* along the segment from ζ, we could materialize the edge for $\mathtt{c}_2$`->prev`.

The key observation we make to define the backward unfolding operation is that we can split segments into subsegments. For example, we can split a $*\!\!=^{i+1}$ segments into a pair of subsegments: $*\!\!=^{i}$ and $*\!\!=^{1}$. This segment splitting property is captured by the following lemma:

Lemma 4.5 (Splitting inductive segments)**.** *Let*

$$(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) *\!\!=^{i+1} \alpha'.c'(\delta')\,)\ .$$

Then, there exists a checker c'' and nodes α'', δ'' such that

$$(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \mathrel{*\!\!=}^{i} \alpha''.c''(\delta'') * \alpha''.c''(\delta'') \mathrel{*\!\!=}^{1} \alpha'.c'(\delta')\,)\ .$$

Proof. By induction on i. Note that α'', δ'' are fresh and that only checkers that may be called transitively from c need to be considered for c''. ☐

Observe that Lemma 4.5 makes it possible to decompose a segment into a *finite* set of disjuncts with shorter segments. We can then define a backward unfolding operation by first splitting an $(i+1)$-step segment into an i-step segment and a 1-step segment and then apply forward unfolding to the 1-step segment (while separately considering the 0-step segment case).

More precisely, we define a *backward unfolding* function $\mathbf{unfold}^{\leftarrow 1}_{\alpha'}$, which should be applied at a node α' in an abstract state of the form $\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle$ and conceptually unfolds the checker application just before α' in the sequence of calls from α to α'.

Definition 4.6 (Backward unfolding of segments)**.**

$$\begin{aligned}
&\mathbf{unfold}^{\leftarrow 1}_{\alpha'}(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \\
&\stackrel{\text{def}}{=}\ \mathbf{unfold}^{0}_{\alpha}(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \lor \\
&\qquad\qquad \left(\bigvee_{c''} \mathbf{unfold}_{\alpha''}(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha''.c''(\delta'') * \alpha''.c''(\delta'') \mathrel{*\!\!=}^{1} \alpha'.c'(\delta'), P\rangle) \right)
\end{aligned}$$

where c'' is over each possible checker and α'', δ'' are fresh (as in Lemma 4.5).

The first disjunct corresponds to the empty segment, while the remaining disjuncts come from splitting the non-empty segment and applying the forward unfolding on the $\mathrel{*\!\!=}^{1}$ edge.

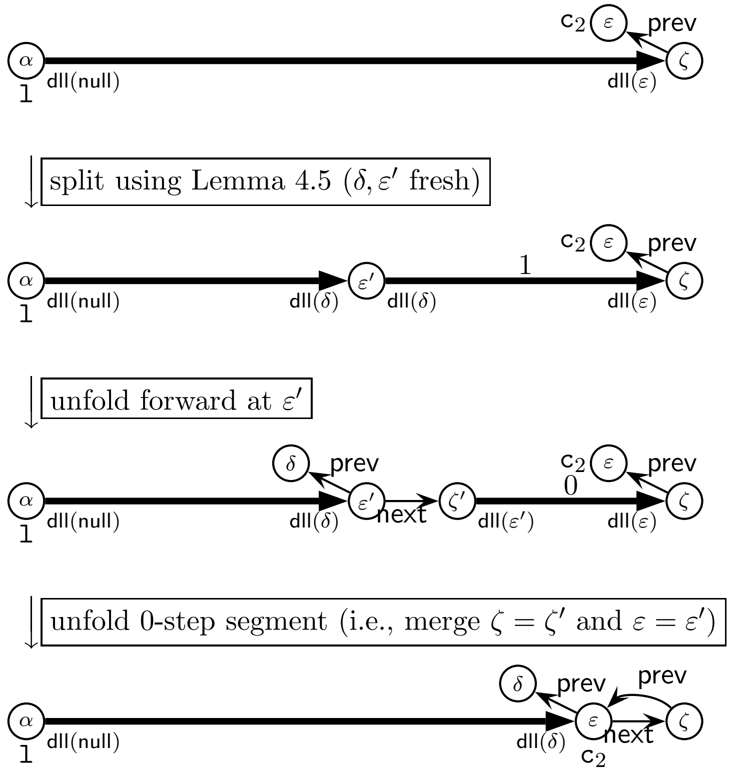


Figure 4.2: Backward unfolding of a doubly-linked list segment. This example shows in detail the backward unfolding in Figure 4.1 of the dll segment from program point 11 to point 12. The backward unfolding operation is broken down into individual steps that demonstrate how it is defined in terms of forward unfolding.

Note that Lemma 4.5 can be generalized to splitting segments of length $i + k$, which allows us to unfold backward k steps in one operation (for any constant k). I write $\mathbf{unfold}_{\alpha'}^{\leftarrow k}$ for the k-step backward unfolding function at α'.

Example. In Figure 4.2, I show the individual steps in the backward unfolding of a doubly-linked list segment that is needed in the example shown in Figure 4.1. At the top, I show the subgraph of interest from program point 11. The empty segment case is ruled out because we have that $\varepsilon \neq \mathsf{null}$ from the loop condition; since the parameter at α is null, an empty segment would imply that $\varepsilon = \mathsf{null}$ (as ε is the parameter at ζ). Figure 4.2

shows the steps for the non-empty segment case. It is in the last step that we discover that $\zeta = \zeta'$ and $\varepsilon = \varepsilon'$, which allows us to find out that ε@$\mathsf{next} \mapsto \zeta$ (i.e., is the points-to edge for `c`$_2$`->next`).

Properties of Unfolding

The definitions in this subsection specify the forward and backward unfolding operations on complete checker edges and segment edges. These operations are sound in that they result in a weaker disjunction.

Theorem 4.7 (Soundness of unfolding)**.**

1. *If* $\mathbf{unfold}_{\alpha}(\langle M * \alpha.c(\delta), P\rangle) = \bigvee_i \langle M_i', P_i'\rangle$,
   then $\boldsymbol{\gamma}(\langle M * \alpha.c(\delta), P\rangle) \subseteq \bigcup_i \boldsymbol{\gamma}(\langle M_i', P_i'\rangle)$.
2. *If* $\mathbf{unfold}_{\alpha}(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) = \bigvee_i \langle M_i', P_i'\rangle$,
   then $\boldsymbol{\gamma}(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \subseteq \bigcup_i \boldsymbol{\gamma}(\langle M_i', P_i'\rangle)$.
3. *If* $\mathbf{unfold}^{\leftarrow 1}_{\alpha'}(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) = \bigvee_i \langle M_i', P_i'\rangle$,
   then $\boldsymbol{\gamma}(\langle M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'), P\rangle) \subseteq \bigcup_i \boldsymbol{\gamma}(\langle M_i', P_i'\rangle)$.

In other words,

if unfolding transforms $\langle M, P\rangle$ *into* $\bigvee_i \langle M_i', P_i'\rangle$, *then* $\boldsymbol{\gamma}(\langle M, P\rangle) \subseteq \bigcup_i \boldsymbol{\gamma}(\langle M_i', P_i'\rangle)$.

There are similar statements for the shape unfolding $\mathbf{u}_{\alpha}$.

Proof. Unfolding of checker calls (fact 1) is proven by induction on the height of the checker "call tree". Similarly, forward unfolding of segments (fact 2) is proven by induction on the length of the segment. For backward unfolding of segments (fact 3), we apply splitting of segments (Lemma 4.5) and the soundness of forward segment unfolding. $\square$

Furthermore, the shape unfolding $\mathbf{u}_\alpha$ is exact in that each rule is considered and all pure constraints are kept.

Lemma 4.8 (Completeness of shape unfolding)**.**

1. *If* $(M', F') \in \mathbf{u}_\alpha(M * \alpha.c(\delta))$, $(\sigma, \nu) \in \boldsymbol{\gamma}(M')$, *and* ν *satisfies* F', *then* $(\sigma, \nu) \in \boldsymbol{\gamma}(M * \alpha.c(\delta))$.
2. *If* $(M', F') \in \mathbf{u}_\alpha(M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'))$, $(\sigma, \nu) \in \boldsymbol{\gamma}(M')$, *and* ν *satisfies* F', *then* $(\sigma, \nu) \in \boldsymbol{\gamma}(M * \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta'))$.

Proof. By the definitions of $\boldsymbol{\gamma}$ (Definition 3.2), and $\models$ (Figure 3.7). $\square$

4.1.2 Expression Evaluation and Controlling Unfolding

The basic transfer functions for atomic commands (e.g., mutation, allocation, and deallocation) are all fairly straightforward, as updates affect graphs locally. As first mentioned in Section 2.4, once points-to edges have been materialized, pointer updates amount to the swinging of an edge. Determining which edge to swing and to where is a simple walk of the graph from variables following the sequence of field dereferences of the command. This update is sound because each edge is a disjoint region of memory (i.e., the separation constraint). Rules specifying the update on graphs for atomic commands are given in Figure 4.3(b). For **new** and **free**, I write $\alpha@\vec{f} \mapsto \vec{\beta}$ as a shorthand for a series of points-to relations for each field (i.e., $\alpha@f_0 \mapsto \beta_0 * \alpha@f_1 \mapsto \beta_1 * \cdots * \alpha@f_n \mapsto \beta_n$). The last rule is the so-called frame rule from separation logic that captures the idea that updates affect graphs locally. That is, because of the disjointness of regions, the only cells that need to be considered are those mentioned in *cmd*. I write lowercase m to connote a fragment of the global memory state. Updating the overall analysis state is described in Figure 4.3(c).

program commands $cmd ::= \mathbf{skip} \mid exp_0 \,.\, f \texttt{=} exp_1 \mid exp \,.\, g \texttt{=} \mathbf{new}(\vec{f}) \mid \mathbf{free}(exp) \mid \cdots$

program expressions $exp ::= x \mid \alpha \mid exp \,.\, f \mid \texttt{-}exp \mid \cdots$

(a) An imperative programming language.

$\{ M \}\ cmd\ \{ M' \}$	Precondition M updated by cmd yields postcondition M'.

$$\frac{}{\{ \mathsf{emp} \}\ \mathbf{skip}\ \{ \mathsf{emp} \}} \qquad \frac{}{\{ \alpha@f \mapsto \beta \}\ \alpha \,.\, f \texttt{=} \gamma\ \{ \alpha@f \mapsto \gamma \}}$$

$$\frac{(\gamma, \vec{\delta} \text{ fresh})}{\{ \alpha@g \mapsto \beta \}\ \alpha \,.\, g \texttt{=} \mathbf{new}(\vec{f})\ \{ \alpha@g \mapsto \gamma * \gamma@\vec{f} \mapsto \vec{\delta} \}} \qquad \frac{}{\{ \alpha@\vec{f} \mapsto \vec{\beta} \}\ \mathbf{free}(\alpha)\ \{ \mathsf{emp} \}}$$

$$\frac{\{ m \}\ cmd\ \{ m' \} \quad (\text{pointers modified by } cmd \text{ not in } M)}{\{ M * m \}\ cmd\ \{ M * m' \}}$$

(b) Updating the abstract memory state.

$\{ A \}\ cmd\ \{ A' \}$	Precondition A updated by cmd yields postcondition A'.

$$\frac{\begin{array}{ll}\mathbf{eval}_E(\langle M, P \rangle,\ exp) = \{\ ..., (\langle M_i, P_i' \rangle,\ \alpha_i), ...\} & \\ \{ M_i \}\ [\alpha_i/exp]cmd\ \{ M_i' \} & (\text{for all } i \text{ and each } exp \text{ in } cmd)\end{array}}{\{ \langle E, M, P \rangle \}\ cmd\ \{ \bigvee_i \langle E, M_i', P_i' \rangle \}}$$

$$\frac{\{ A_0 \}\ cmd\ \{ A_0' \} \quad \{ A_1 \}\ cmd\ \{ A_1' \}}{\{ A_0 \vee A_1 \}\ cmd\ \{ A_0' \vee A_1' \}}$$

(c) Updating the analysis state.

Figure 4.3: Transforming the analysis state.

$\boldsymbol{eval}_E(\langle M, P\rangle,\ exp) = \{\ ..., (\langle M_i, P_i\rangle,\ \alpha_i), ...\}$	$\langle M, P\rangle$ unfolds to $\bigvee_i \langle M_i, P_i\rangle$ so that *exp* evaluates to α_i in M_i.

$$\frac{}{\boldsymbol{eval}_E(\langle M, P\rangle,\ x) = \{\ (\langle M, P\rangle,\ E(x))\ \}}$$

$$\frac{\begin{array}{l}\boldsymbol{eval}_E(\langle M, P\rangle,\ exp) = \{\ ..., (\langle M_i, P_i\rangle,\ \alpha_i), ...\} \\ \mathbf{unfold}_{\alpha_i @ f}(\langle M_i, P_i\rangle) = \bigvee_j \langle M_j * \alpha_i @ f \mapsto \beta_j, P_j\rangle \quad \text{(for all } i, j)\end{array}}{\boldsymbol{eval}_E(\langle M, P\rangle,\ exp\,.\,f) = \{\ ..., (\langle M_j * \alpha_i @ f \mapsto \beta_j, P_j\rangle,\ \beta_j), ...\}}$$

$$\frac{\boldsymbol{eval}_E(\langle M, P\rangle,\ exp) = \{\ ..., (\langle M_i, P_i\rangle,\ \alpha), ...\} \quad (\beta \text{ fresh})}{\boldsymbol{eval}_E(\langle M, P\rangle,\ \texttt{-}exp) = \{\ ..., (\langle M_i, \mathbf{guard}_{\mathbb{P}^\sharp}(P_i, \beta = -\alpha)\rangle,\ \beta), ...\}}$$

Figure 4.4: Symbolically evaluating program expressions in abstract memory states.

The $\boldsymbol{eval}$ function captures the notion of walking a graph according to an expression *exp* to obtain a symbolic value α. To obtain a symbolic value, it may need to apply unfolding (i.e., materialize the needed points-to edges). The first rule simply states that to reflect a command *cmd* in the analysis state, first the graph is unfolded so that we have a node for each expression *exp* in *cmd*, and then each graph is updated[1]. Observe that performing this symbolic evaluation may update the pure state P (e.g., because of unfolding). Also, note that since $\boldsymbol{eval}$ uses the unfolding operations, the local soundness of this rule relies on the soundness of unfolding (Theorem 4.7). Finally, overall, for a disjunctive analysis state, it simply transforms each disjunct individually.

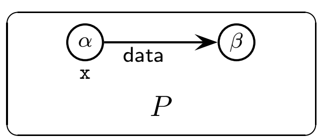
Propagating Data Constraints

The $\boldsymbol{eval}$ function is inductively defined in Figure 4.4. First, variables evaluate to their addresses as given by the environment E. In the second rule, we use an unfolding

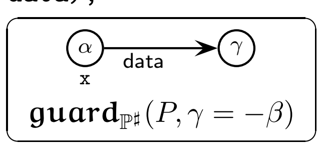
[1]For presentation purposes, the statement of this rule is slightly informal, as the intent is that $\boldsymbol{eval}$ is done simultaneously for each expression in *cmd* (specifically, for the mutation command).

operation to materialize a field. I write $\mathbf{unfold}_{\alpha@f}$ to mean apply whatever unfolding operation to try to materialize the cell with address $\alpha@f$. For data operations (e.g., arithmetic expressions) on heap values, we symbolically evaluate the expressions by obtaining symbolic values for each memory access. We then create a new symbolic value to stand for this expression in the graph and assert this equality relation in the data domain $\mathbb{P}^\sharp$. The last rule shows one such case. Note that this constraint propagation step is how the shape domain extends the base data domain to reason about heap values. The base data domain should also have a variable elimination operation so that the shape domain can inform it when certain nodes are no longer relevant (e.g., because regions get summarized).

Example 4.4 (Propagating data constraints)**.** The following mutation command shows an example transition that propagates data constraints during evaluation of the program expression.




```
x->data = -(x->data);
```



Deciding Where to Unfold

As described above, evaluating expression requires following points-to edges in the shape graph. In case the relevant points-to edges are folded into complete or partial checker edges (i.e., summarized), we need to unfold the appropriate checker edge to materialize the desired points-to edge (i.e., the $\mathbf{unfold}_{\alpha@f}$ function from above). To choose the appropriate edge and unfolding operation, we take advantage of the type inference on checker parameters defined in Section 3.4.

We are faced with deciding where to perform unfolding when the evaluation of an expression requires dereferencing a field f of a node α, but there is no such points-to edge from α. If there is a complete checker edge $\alpha.c(\delta)$ or a partial checker edge $\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$ starting from α (i.e., α is the traversal argument for some checker edge), then we can unfold this summary edge using the forward unfolding function $\mathbf{unfold}_{\alpha}$. If the points-to edge for $\alpha@f$ is materialized, then the evaluation of the expression can be resumed in the new unfolded graph(s). This materialization step is the basic one based on the knowledge that points-to edges emanate from the traversal parameter in inductive checker definitions and is what applies at program point 3 and program point 7 in Figure 4.1. Note that as an optimization, we need only consider outgoing checker edges where the type of the traversal parameter of the checker includes $f\langle\ell\rangle$ (for a level ℓ that is non-negative or unk).

Otherwise, if there is no outgoing checker edge from α, we look for a potential backward unfolding. We look elsewhere for a partial checker edge $\beta.c(\delta) \mathrel{*\!\!=} \beta'.c'(\alpha)$ where α is a parameter at the tail. If additionally, the corresponding parameter of checker c' has a type that includes $f\langle n\rangle$ where $n < 0$, then we apply the backward unfold function at β' ($\mathbf{unfold}^{\leftarrow|n|}_{\beta'}$). The magnitude of the integer level tells us how many steps backward (i.e., how to split the segment from β to β'). In the doubly-linked list example (Figure 4.1) at program point 11, we are trying to materialize $\varepsilon@\mathsf{prev}$ when ε has no outgoing edges. However, we have the edge $\alpha.\mathsf{dll}(\mathsf{null}) \mathrel{*\!\!=} \zeta.\mathsf{dll}(\varepsilon)$ in the graph. Since the type of the additional parameter to $\mathtt{dll}$ contains $\mathtt{prev}\langle\mathtt{-1}\rangle$ (see the type of $\mathtt{l}_{\mathrm{p}}$ in Example 3.9), we know to unfold backward 1-step from ζ. Observe that the checker parameter typing does not affect soundness; it is utilized only as guide to decide where to unfold.

Unfolding and Reduction

Often, all but one of the disjuncts are ruled out during unfolding by contradictory data constraints. For example, a conditional guard that gives $\alpha \neq \mathsf{null}$ rules out the empty case when, for example, unfolding a doubly-linked list checker edge $\alpha.\mathsf{dll}(\mathsf{null})$ (the checker from Example 3.1, Example 3.5, and Example 3.7). In our discussion, we assume that all relevant information is shared between the shape domain and the data domain (i.e., we maintain a fully reduced representation). In implementation, we must decide what kind of constraints derived by one domain should be propagated to the other so that we can quickly rule out contradictory states. In the shape domain, the disjointness of memory regions gives implicitly a disequality between pointers. In particular,

if $\alpha @ f \mapsto \beta * \gamma @ f \mapsto \delta$, *then* $\alpha \neq \gamma$.

Conversely, if the data domain derives an equality constraint between pointers, then conveying that information to the shape domain allows it to merge nodes and detect such contradictions.

4.2 History-Guided Folding with Data Constraints

To obtain loop invariants in the shape domain, we need a way to identify subgraphs that should be *folded* into complete or partial checker edges. What kinds of subgraphs can be summarized without losing too much precision is highly dependent on the structures in question and the code being analyzed. To see this, consider the fixed-point graph at program point 4 in this skip list example from Figure 2.5. One could imagine folding the points-to edges corresponding to `p->n` and `p->s` into one summary region from `p` to `c` (i.e., eliminating the node labeled γ), but it is necessary to retain the information that `p` and `c` are "separated" by *at least one* `next` field. Keeping node γ expresses this fact. Rather than using a *canonicalization* operation that looks only at one graph to identify the subgraphs that should be summarized, our weakening strategy is based on the observation that previous iterates at loop join points can be utilized to guide the folding process. Furthermore, because checker edges incorporate both shape and data properties, this summarization requires careful coordination between the shape domain and the data domain (in order to avoid losing precision unnecessarily), particularly in the presence of relational checkers.

In this subsection, I define the *comparison* and *widening* operations, which both first perform a simultaneous traversal over the input shape graphs gathering constraints before then applying the corresponding operation in the base domain. I describe the comparison algorithm first, as it has similar but slightly simpler structure as compared to the widening and is also the key subroutine used by the widening.

4.2.1 Comparison of Analysis States

The comparison operator checks inclusion between two abstract elements in a conservative way. More precisely, it takes as input two abstract elements

$$A_\ell = \langle E_\ell, M_\ell, P_\ell \rangle \qquad \text{and} \qquad A_r = \langle E_r, M_r, P_r \rangle$$

and returns `true` if it can establish that

$$\boldsymbol{\gamma}(A_\ell) \subseteq \boldsymbol{\gamma}(A_r)$$

and false otherwise (which does not necessarily mean that the inclusion does not hold at the concrete level). Static analyses rely on the comparison operator in order to ensure the termination of fixed-point computation. We also utilize it to collapse extraneous disjuncts in the analysis state and most importantly, as a subroutine in the widening operation.

Recall that the nodes in the shape graph correspond to existentially-quantified symbolic values, so at the basis of the comparison is a notion of node equivalence, which states that valuations should map nodes in A_ℓ and A_r to the *same* value for the inclusion to hold. For instance, if x is a variable, then the address of x should be the same on both sides, or the inclusion *cannot* hold. In fact, these equality relations constrain the valuations. Thus, when it succeeds, the comparison algorithm should return a *valuation transformer* Ψ that is a function mapping nodes of A_r into nodes of A_ℓ. The condition that Ψ is a function ensures that any aliasing expressed in A_r is also reflected in A_ℓ, so if at any point, this condition on Ψ is violated, then the comparison returns false.

At a high-level, the algorithm proceeds in three stages:

- First, the *initialization* of the algorithm creates an initial valuation transformer Ψ_{init} defined by the environments E_ℓ and E_r. Each variable should be mapped to the same address, so it is defined as follows:

  $$\forall x \in \mathbf{Var}, \Psi_{\text{init}}(E_r(x)) = E_\ell(x) \ .$$

  The valuation transformer plays an essential role in the algorithm itself since it gives the points from where we should compare the graphs. It is initialized using the environment, as the natural starting points are the nodes that correspond to the program variables.

- Second, a *comparison in the shape domain* is performed, which proceeds by checking inclusion locally. When new node relations are established as required for the inclusion to hold, the valuation transformer should be extended in order to include these constraints. Finally, it returns the following:

  1. the final valuation transformer Ψ;
  2. a first-order formula F, which collects pure constraints, which may arise during the computation that must ultimately be proven (i.e., are temporarily assumed).

- Last, a *comparison in the data domain* is performed that shows the inclusion of P_ℓ in P_r. We must also ask the data domain to prove and discharge the first-order side-conditions F computed in the previous step hold under the assumption of P_ℓ. All this is done modulo application of the valuation transformer Ψ.

Comparison in the Shape Domain

The basic idea of the graph comparison algorithm is to determine semantic inclusion by iteratively reducing to stronger statements until the inclusion is obvious. It does so using a set of rules that apply to the graphs locally. While applying this set of rules, the algorithm carries along and enriches the pair (Ψ, F) introduced above.

The rules are presented in Figure 4.5. For conciseness, I omit the explicit bookkeeping of the node relations in the valuation transformer Ψ, that is, the rules assume the "final" Ψ is given so that a derivation states the soundness of the whole computation. In practice, the state of Ψ also determines when a rule applies. I show this aspect indirectly by underlining the constraints on Ψ that are added once the rule is used, while the mappings

$M_\ell \sqsubseteq_\Psi^{F_\ell} M_r$	M_ℓ is approximated by M_r under Ψ and with residual conditions F.

$$\frac{}{\mathsf{emp} \sqsubseteq_\Psi^{\mathsf{t}} \mathsf{emp}}\ \textsf{c-emp} \qquad \frac{\Psi(\alpha_r) = \alpha_\ell \qquad M_\ell \sqsubseteq_\Psi^F M_r \qquad \underline{\Psi(\beta_r) = \beta_\ell}}{M_\ell * \alpha_\ell @ f \mapsto \beta_\ell \sqsubseteq_\Psi^F M_r * \alpha_r @ f \mapsto \beta_r}\ \textsf{c-pt}$$

$$\frac{\Psi(\alpha_r) = \alpha_\ell \qquad M_\ell \sqsubseteq_\Psi^F M_r \qquad \underline{\Psi(\delta_r) = \delta_\ell}}{M_\ell * \alpha_\ell.c(\delta_\ell) \sqsubseteq_\Psi^F M_r * \alpha_r.c(\delta_r)}\ \textsf{c-chk}$$

$$\frac{\begin{array}{l} \Psi(\alpha_r) = \alpha_\ell \hfill M_\ell \sqsubseteq_\Psi^F M_r * \alpha'_r.c'(\delta'_r) \\ \underline{\Psi(\delta_r) = \delta_\ell} \qquad \underline{\Psi(\alpha'_r) = \alpha'_\ell} \qquad \underline{\Psi(\delta'_r) = \delta'_\ell} \qquad (\alpha'_r, \delta'_r \text{ fresh}) \end{array}}{M_\ell * \alpha_\ell.c(\delta_\ell) \mathrel{*\!\!=} \alpha'_\ell.c'(\delta'_\ell) \sqsubseteq_\Psi^F M_r * \alpha_r.c(\delta_r)}\ \textsf{c-segchk}$$

$$\frac{\begin{array}{l} \Psi(\alpha_r) = \alpha_\ell \hfill M_\ell \sqsubseteq_\Psi^F M_r * \alpha'_r.c'(\delta'_r) \mathrel{*\!\!=} \alpha''_r.c''(\delta''_r) \\ \underline{\Psi(\delta_r) = \delta_\ell} \qquad \underline{\Psi(\alpha'_r) = \alpha'_\ell} \qquad \underline{\Psi(\delta'_r) = \delta'_\ell} \qquad (\alpha'_r, \delta'_r \text{ fresh}) \end{array}}{M_\ell * \alpha_\ell.c(\delta_\ell) \mathrel{*\!\!=} \alpha'_\ell.c'(\delta'_\ell) \sqsubseteq_\Psi^F M_r * \alpha_r.c(\delta_r) \mathrel{*\!\!=} \alpha''_r.c''(\delta''_r)}\ \textsf{c-segseg}$$

$$\frac{M_\ell \sqsubseteq_\Psi^F M'_r \qquad (M'_r, F') \in \mathbf{u}_{\alpha_r}(M_r * \alpha_r.c(\delta_r))}{M_\ell \sqsubseteq_\Psi^{F \wedge F'} M_r * \alpha_r.c(\delta_r)}\ \textsf{c-uchk}$$

$$\frac{M_\ell \sqsubseteq_\Psi^F M'_r \qquad (M'_r, F') \in \mathbf{u}_{\alpha_r}(M_r * \alpha_r.c(\delta_r) \mathrel{*\!\!=} \alpha'_r.c'(\delta'_r))}{M_\ell \sqsubseteq_\Psi^{F \wedge F'} M_r * \alpha_r.c(\delta_r) \mathrel{*\!\!=} \alpha'_r.c'(\delta'_r)}\ \textsf{c-useg}$$

Figure 4.5: The comparison operation in the shape domain.

that are not underlined must be in Ψ for the rule to apply. For instance, rule $\mathsf{c\text{-}pt}$ applies when α_ℓ and α_r match (i.e., when $\Psi(\alpha_r) = \alpha_\ell$) and when there is a field edge with label f from each node in both graphs. Then, the edges can be removed from both abstract elements (since $\alpha_\ell @ f \mapsto \beta_\ell$ is obviously weaker than $\alpha_r @ f \mapsto \beta_r$). A correspondence between β_ℓ and β_r can be added into Ψ, for these two nodes should correspond to the same value. When adding such a correspondence is not possible, because it would make Ψ not a function, the algorithm should return false (i.e., the inclusion cannot be established because this situation would mean that one value in M_r should be equal to two possibly distinct values in M_ℓ).

In the following, I briefly summarize the behavior of the other rules, whereas I show example derivations to give a more intuitive understanding of how the rules apply in Example 4.5 and Example 4.6. Rule $\mathsf{c\text{-}emp}$ allows returning true when the proof is finished. Similar to $\mathsf{c\text{-}pt}$, rule $\mathsf{c\text{-}chk}$ matches two checker edges from related nodes. When there is a partial checker edge in M_ℓ and a checker edge in M_r, we split out the "prefix segment" in the right to match the left (rule $\mathsf{c\text{-}segchk}$ for a complete checker edge in M_r and rule $\mathsf{c\text{-}segseg}$ for a partial checker edge). Rules $\mathsf{c\text{-}uchk}$ and $\mathsf{c\text{-}useg}$ unfold complete or partial checker edges in M_r when no other rule applies. Intuitively, when the comparison succeeds, Ψ gives us a relationship between the valuation of nodes on the left and on the right. In other words, a valuation ν_ℓ for A_ℓ can be composed with Ψ to give a valuation for A_r. I write this composition as $\nu_\ell \circledcirc \Psi$, which is used to state soundness.

Theorem 4.9 (Soundness of comparison in the shape domain)**.** *The comparison operation on shape graphs is sound.*

If $M_\ell \sqsubseteq_\Psi^F M_r$, $(\sigma, \nu) \in \boldsymbol{\gamma}(M_\ell)$, *and* $\nu \circledcirc \Psi$ *satisfies* F, *then* $(\sigma, \nu \circledcirc \Psi) \in \boldsymbol{\gamma}(M_r)$.

Proof. By induction on the derivation of $M_\ell \sqsubseteq_\Psi^F M_r$. See Theorem A.5. □

The comparison algorithm in the shape domain is incomplete (i.e., the comparison may fail to prove the inclusion when it does hold at the concrete level). For example, we never consider unfolding in M_ℓ. These rules have been primarily designed to be effective in the way the comparison is used in the join and widening algorithms where we need to see if M_ℓ is an unfolded version of M_r.

Comparison in the Combined Domain

If the comparison in the shape domain succeeds, then the comparison holds in the combined domain if we can discharge the side-conditions F and show the inclusion in the data domain. The key is that the comparison in the shape domain has computed Ψ, the correspondence between values in the left and values in the right, which captures the relationship between the shape and data domains.

To define the overall comparison, we assume that the data domain has a function $\boldsymbol{prove}_{\mathbb{P}^\sharp}$ that takes as input an abstract element $P \in \mathbb{P}^\sharp$ and a first-order formula F and tries to prove that any valuation ν in $\boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P)$ satisfies F, as well as a conservative comparison function $\sqsubseteq_{\mathbb{P}^\sharp}$. Furthermore, we assume the data domain can apply $\circledcirc$ at the abstract level, that is, $P \circledcirc \Psi$ applies the valuation transformer Ψ to rename symbolic values and capture any relations. Conceptually, this operation can be implemented by asserting equalities for each mapping in Ψ then projecting out the symbolic values in the range of Ψ (as in Chang and Leino [2005]). Note that Ψ may track relations, so that $P \circledcirc \Psi$ may collect additional constraints, which were not in P (if $\mathbb{P}^\sharp$ can track them). Together, these operations should satisfy the following conditions:

1. *If* $\mathbf{prove}_{\mathbb{P}^\sharp}(P \otimes \Psi, F)$*, then* $\nu \otimes \Psi$ *satisfies* F*.*
2. *If* $P_\ell \otimes \Psi \sqsubseteq_{\mathbb{P}^\sharp} P_r$*, then* $\boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P_\ell \otimes \Psi) \subseteq \boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P_r)$*.*

With that, the comparison function for the product domain is defined as follows:

Definition 4.10 (Comparison in the combined shape and data domain)**.**

$$\begin{array}{l} \langle M_\ell, P_\ell \rangle \sqsubseteq_\Psi \langle M_r, P_r \rangle \\ \quad \textit{iff} \quad \textit{there exists an } F \textit{ such that} \\ \qquad\qquad M_\ell \sqsubseteq_\Psi^F M_r\,,\ \mathbf{prove}_{\mathbb{P}^\sharp}(P_\ell \otimes \Psi, F)\,,\ \textit{and}\ P_\ell \otimes \Psi \sqsubseteq_{\mathbb{P}^\sharp} P_r\,. \end{array}$$

Moreover, $\langle E_\ell, M_\ell, P_\ell \rangle \sqsubseteq \langle E_r, M_r, P_r \rangle$ *if and only if the above comparison evaluates successfully when started with* $\Psi = \Psi_{\text{init}}$*.*

Theorem 4.11 (Soundness of comparison in the combined shape and data domain)**.**

The comparison operation is sound.

$$\textit{If}\ \langle M_\ell, P_\ell \rangle \sqsubseteq_\Psi \langle M_r, P_r \rangle\ \textit{and}\ (\sigma, \nu) \in \boldsymbol{\gamma}(\,\langle M_\ell, P_\ell \rangle\,),\ \textit{then}\ (\sigma, \nu \otimes \Psi) \in \boldsymbol{\gamma}(\,\langle M_r, P_r \rangle\,).$$

As a consequence, if $\langle E_\ell, M_\ell, P_\ell \rangle \sqsubseteq \langle E_r, M_r, P_r \rangle$*, then* $\boldsymbol{\gamma}(\,\langle E_\ell, M_\ell, P_\ell \rangle\,) \subseteq \boldsymbol{\gamma}(\,\langle E_r, M_r, P_r \rangle\,)$*.*

Proof. Direct by the soundness of comparison in the shape domain (Theorem 4.9) and the soundness conditions on $\mathbf{prove}_{\mathbb{P}^\sharp}$ and $\sqsubseteq_{\mathbb{P}^\sharp}$ in the data domain. $\square$

Examples

I now present a few examples of deciding inclusion. In Example 4.5, the focus is on the comparison algorithm in the shape domain (as defined in Figure 4.5). Then, Example 4.6 discusses in further detail the interaction with the base data domain.

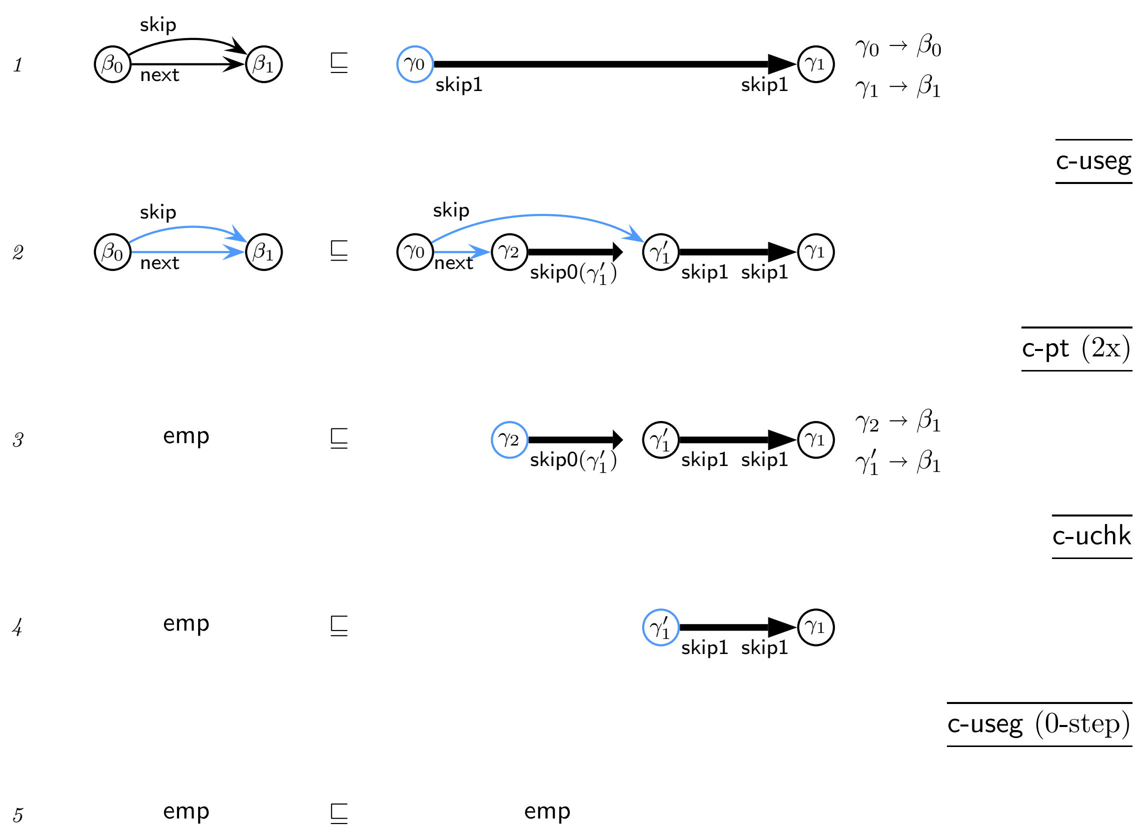


Figure 4.6: A derivation deciding the inclusion of a one-node skip list fragment in a skip list segment of checker skip1 from Figure 2.4(b). The comparison operation works by iteratively reducing to stronger statements (defined in Figure 4.5). This example is described in Example 4.5.

Example 4.5 (A comparison operation on skip lists)**.** I highlight aspects of the comparison algorithm by following an example derivation shown in Figure 4.6 (from goal to axiom). The top line shows the initial goal with a particular initialization for the valuation transformation Ψ. Each subsequent line shows a step in the derivation that is obtained by applying the named rule (shown flush right). The highlighting of nodes and edges indicates where the rules apply. To the right of the graphs, I show the valuation transformer Ψ as it is extended through the course of the computation. We are able to prove that the left graph is included in the right graph because we reach $\mathsf{emp} \sqsubseteq^{\mathsf{t}}_{\Psi} \mathsf{emp}$.

First, consider the application of the c-pt rule (line 2 to 3). When both M_ℓ and M_r have the same kind of edge from matched nodes, the approximation relation obviously holds for those edges, so those edges can be consumed. Any target nodes are then added to Ψ so that the traversal can continue from those nodes. In this case, the skip and next points-to edges match from the node mapping $\gamma_0 \to \beta_0$. With this matching, the mappings $\gamma_2 \to \beta_1, \gamma_1' \to \beta_1$ are added. The c-chk rule is the analogous matching rule for complete checker edges (not shown in this example).

Now, consider the first application of c-useg (line 1 to 2) where we have a segment from γ_0 on the right, but we do not have an edge from β_0 on the left that can be immediately matched with it. In this case, we unfold the segment. In general, unfolding results in a disjunction of graphs, so the overall approximation check succeeds if the approximation check succeeds for any *one* of the unfolded graphs. In the application of c-uchk (line 3 to 4), the unfolding of $\gamma_2.\mathsf{skip0}(\gamma_1')$ is to emp because we have that $\gamma_2 = \gamma_1'$. This equality arises because they are both unified with β_1. (specifically, the c-pt steps added $\gamma_2 \to \beta_1$ and

$\gamma_1' \to \beta_1$ to Ψ). Finally, the last step unfolds the segment γ_1' as a 0-step segment.

The remaining rules not discussed in the above example (c-segchk and c-segseg) concern partial checker edges (i.e., segments). Observe that we apply edge matching only to points-to edges (c-pt) and complete checker edges (c-chk). For better precision, partial checker edges are treated differently. As segments summarize any number of steps, two segments that start at corresponding nodes need not be used to summarize the same number of steps. For example, consider a valuation transformer of $\alpha_r \to \alpha_\ell$, a segment from α_ℓ to α_ℓ'' on the left and then an unfolded region from α_ℓ'' to α_ℓ' (i.e., the "appending" of an unfolded region to a segment), and a segment from α_r to α_r' on the right. Intuitively, this inclusion should hold (provided the unfolded region is described by the segment checker). A "direct segment matching rule" would incorrectly match the two input segments, leaving the unfolded region and causing the comparison to fail; however, application of c-segseg followed by c-useg allows a derivation to be constructed to prove this inclusion. Rules c-segchk and c-segseg can be viewed as iterative unfolding of the segment on the right to "consume" the region on the right (recall, the comparison operation is designed to see if M_ℓ is an unfolded version of M_r). The "direct segment matching rule" can be derived by first applying c-segseg followed by c-useg (0-step case).

Example 4.6 (Verifying a loop invariant of a search tree traversal)**.** In this example, we check, for a region, the inclusion of an iteration in a loop invariant for finding the value d in a binary search tree (essentially, the first stage prior to the insertion in the red-black tree example, as shown in Figure 3.2, lines 2–8). I present the derivation in Figure 4.7.

The first line shows the initial goal: on the left-side of the comparison, we have

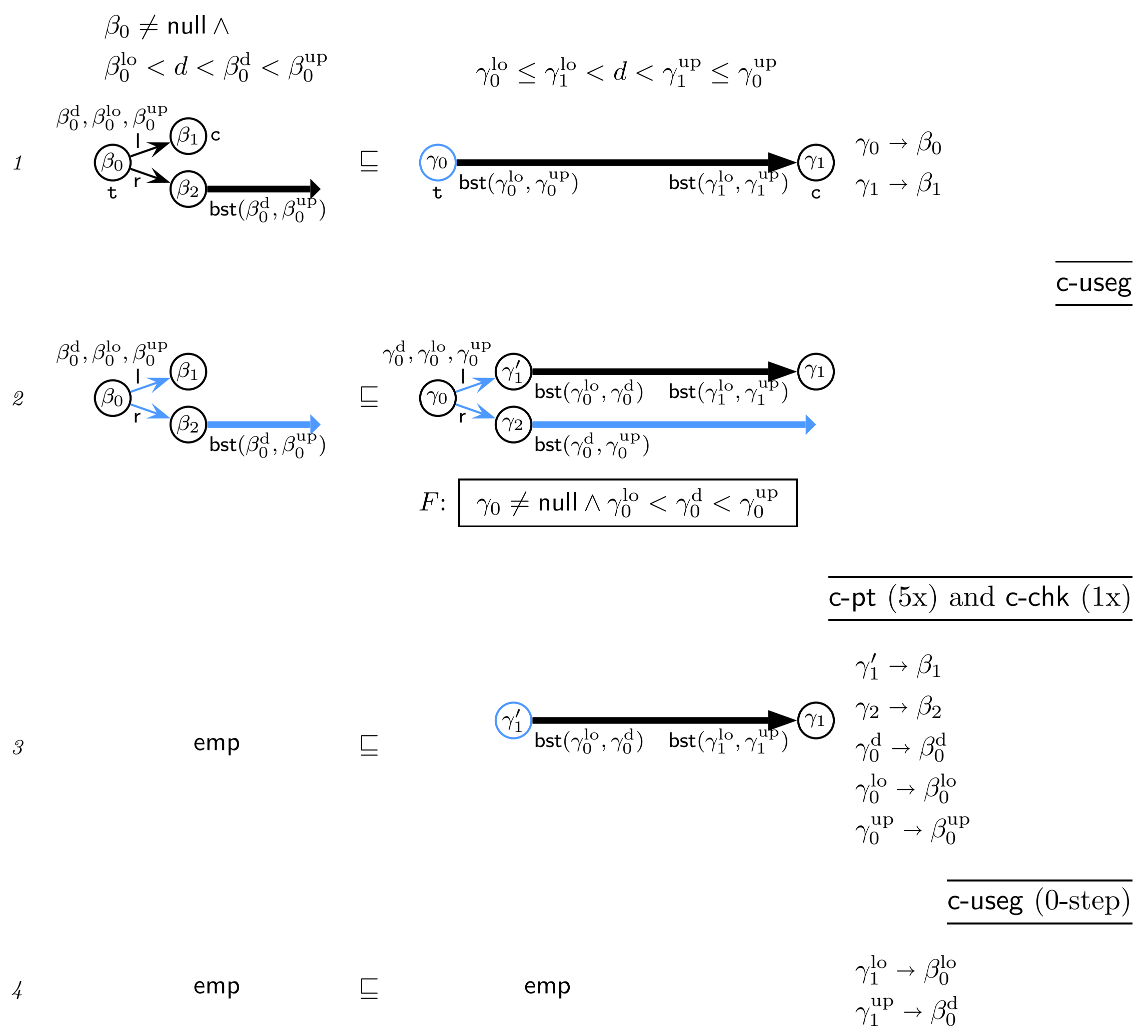


Figure 4.7: A derivation deciding the inclusion of a search tree fragment with at least one node in a search tree segment of checker bst from Example 3.2. The example shows the core of verifying a loop invariant of a search tree traversal and is described in Example 4.6.

the state where the cursor `c` has advanced to the left subtree of the root `t`. We want to show that this subgraph is included in the segment from `t` to `c`. At the top, we show the pure constraints for each side: on the left, we have that $d < \beta_0^{\mathrm{d}}$, which is why `c` advanced to the left subtree. We want to show that d is in the range of the subtree from γ_1. To keep the diagram compact, I write the values of the data fields as a tuple (e.g., $\beta_0^{\mathrm{d}}, \beta_0^{\mathrm{lo}}, \beta_0^{\mathrm{up}}$).

The first step (line 1 to 2) applies $\mathsf{c\text{-}useg}$ that unfolds the segment on the right producing the following proof obligation F (shown boxed in the picture above):

$$\gamma_0 \neq \mathsf{null} \;\wedge\; \gamma_0^{\mathrm{lo}} < \gamma_0^{\mathrm{d}} < \gamma_0^{\mathrm{up}} \; .$$

The next step (line 2 to 3) matches points-to and complete checker edges, which extends Ψ. Finally, the last step unfolds the segment at γ_1' as a 0-step segment, which produces key additional constraints on Ψ that come from the semantics of the 0-step segment.

To complete the proof, we need to discharge the above proof obligation and show inclusion at the data level. Applying the valuation transformer to the element of the data domain on the left side (i.e., $P_\ell \circledcirc \Psi$), we get the following:

$$\gamma_0 \neq \mathsf{null} \;\wedge\; \gamma_0^{\mathrm{lo}} = \gamma_1^{\mathrm{lo}} < d < \gamma_0^{\mathrm{d}} = \gamma_1^{\mathrm{up}} < \gamma_0^{\mathrm{up}}$$

which implies the proof obligation and the inequality constraints on the right side (i.e., the loop invariant for the data).

4.2.2 Join and Widening of Analysis States

The join and widening operators combine shape and data constraints to build an upper bound (i.e., an over-approximation) of two abstract elements

$$A_\ell = \langle E_\ell, M_\ell, P_\ell \rangle \qquad \text{and} \qquad A_r = \langle E_r, M_r, P_r \rangle \; .$$

Furthermore, the widening operation should ensure the termination of sequences of abstract iterates. In particular, termination should be achieved at both the shape and data levels.

We first consider the join of abstract elements, that is, the computation of a sound approximation of both A_ℓ and A_r. Like for the comparison operator, we need to track the correspondence between symbolic values in the inputs and those in the output. Intuitively, a node α in the result should over-approximate the values corresponding to a pair of nodes (α_ℓ, α_r), where α_ℓ is in A_ℓ and α_r in A_r, so we maintain a pair of valuation transformers (Ψ_ℓ, Ψ_r) that describe these relations. These valuation transformers should be consistent with the environments and need to be used when joining the abstractions of data P_ℓ and P_r. For convenience, I write $\Psi(\alpha)$ for $(\Psi_\ell(\alpha), \Psi_r(\alpha))$.

At a high-level, we can partition the join into stages in a similar manner as the comparison operation (by utilizing the valuation transformers Ψ_ℓ, Ψ_r):

- First, during *initialization*, for each variable x in the environment, a node α_x is created so as to represent the address of x. The valuation transformers Ψ_ℓ, Ψ_r are initialized so that

$$\forall x \in \mathbf{Var},\ \Psi_{\text{init}}(\alpha_x) = (E_\ell(x), E_r(x))\ ,$$

  and the resulting environment E is defined by $\forall x \in \mathbf{Var},\ E(x) = \alpha_x$.

- Second, a *join in the shape domain* builds a new shape abstraction M and returns it together with valuation transformers Ψ_ℓ, Ψ_r and residual first-order constraints F_ℓ, F_r that should be proven at the data level. Like in the comparison, it also enriches Ψ_ℓ and Ψ_r whenever a new node is created in order to preserve the consistency of the node pairing.

- Last, a *join in the data domain* is applied to $P_\ell \circledast \Psi_\ell$ and $P_r \circledast \Psi_r$. We must also ask the data domain to discharge the first-order constraints F_ℓ, F_r (so as to check that the abstractions performed in the shape join are valid with respect to data constraints).

Note that while the join requires a cooperation between the shape domain and data domain (as checker edges include data constraints), we are able to get a clean separation in phases because of the first-order constraints F_ℓ, F_r and the valuation transformers Ψ_ℓ, Ψ_r. In essence, the shape phase identifies regions that should be folded with respect to shape assuming the data constraints are satisfied. Then, the data phase guarantees that the constraints are indeed satisfied.

Join in the Shape Domain

The join process in the shape domain is similar to the comparison in that we do a simultaneous traversal over the input graphs guided by the valuation transformers Ψ_ℓ, Ψ_r. The basic idea is to perform rewritings based on the following derived rule of inference in separation logic:

$$\frac{m_\ell \Rightarrow m \quad m_r \Rightarrow m}{(M_\ell * m_\ell) \vee (M_r * m_r) \Rightarrow (M_\ell \vee M_r) * m}$$

In terms of an algorithm, the join iteratively attempts to replace fragments in each of the input shape graphs (m_ℓ and m_r) with a new fragment (m) through a set of rewriting rules. A rule "consumes" fragments m_ℓ of M_ℓ and m_r of M_r and produces a fragment m for the result, which should be a sound approximation of m_ℓ and m_r modulo the application of Ψ_ℓ and Ψ_r, respectively.

$\langle m_\ell, m_r \rangle \rightsquigarrow_{\Psi}^{F_\ell, F_r} m$	Under Ψ, m_ℓ and m_r are approximated by m with residual conditions F_ℓ, F_r.

$$\frac{\Psi(\alpha) = (\alpha_\ell, \alpha_r) \qquad \underline{\Psi(\beta) = (\beta_\ell, \beta_r)}}{\langle \alpha_\ell @ f \mapsto \beta_\ell, \alpha_r @ f \mapsto \beta_r \rangle \rightsquigarrow_{\Psi}^{\mathsf{t},\mathsf{t}} \alpha @ f \mapsto \beta}\ \textsf{j-pt} \qquad \frac{\Psi(\alpha) = (\alpha_\ell, \alpha_r) \qquad \underline{\Psi(\delta) = (\delta_\ell, \delta_r)}}{\langle \alpha_\ell.c(\delta_\ell), \alpha_r.c(\delta_r) \rangle \rightsquigarrow_{\Psi}^{\mathsf{t},\mathsf{t}} \alpha.c(\delta)}\ \textsf{j-chk}$$

$$\frac{\Psi(\alpha) = (\alpha_\ell, \alpha_r) \qquad m_r \sqsubseteq_{\Psi_r}^{F} \alpha.c(\delta) \qquad \underline{\Psi_\ell(\delta) = \delta_\ell}}{\langle \alpha_\ell.c(\delta_\ell), m_r \rangle \rightsquigarrow_{\Psi}^{\mathsf{t},F} \alpha.c(\delta)}\ \textsf{j-wchk}$$

$$\frac{\begin{array}{lll}\Psi(\alpha) = (\alpha_\ell, \alpha_r) & \Psi(\alpha') = (\alpha'_\ell, \alpha'_r) & m_r \sqsubseteq_{\Psi_r}^{F} \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta') \\ \underline{\Psi_\ell(\delta) = \delta_\ell} & \underline{\Psi_\ell(\delta') = \delta'_\ell} & \end{array}}{\langle \alpha_\ell.c(\delta_\ell) \mathrel{*\!\!=} \alpha'_\ell.c'(\delta'_\ell), m_r \rangle \rightsquigarrow_{\Psi}^{\mathsf{t},F} \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')}\ \textsf{j-wseg}$$

$$\frac{\begin{array}{lll}\Psi(\alpha) = (\alpha_\ell, \alpha_r) & \Psi(\alpha') = (\alpha_\ell, \alpha'_r) & m_r \sqsubseteq_{\Psi_r}^{F} \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c(\delta') \\ \underline{\Psi_\ell(\delta) = \delta_\ell} & \underline{\Psi_\ell(\delta') = \delta_\ell} & \end{array}}{\langle \mathsf{emp}, m_r \rangle \rightsquigarrow_{\Psi}^{\mathsf{t},F} \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c(\delta')}\ \textsf{j-waliases}$$

Figure 4.8: Fragment rewriting rules for the join operation in the shape domain.

In Figure 4.8, I present the fragment rewriting rules. I write

$$\langle m_\ell, m_r \rangle \rightsquigarrow_{\Psi}^{F_\ell, F_r} m$$

for such a rewriting rule where Ψ is the valuation transformer and F_ℓ, F_r are the residual first-order constraint from the rewriting. Like for the comparison, I do not explicitly show the extending of Ψ but rather assume the "final" Ψ is given. Also, the rules for the join are intended to be symmetric; for conciseness, I elide the left-sided version of the non-symmetric right-sided rules.

As alluded to above, the basic idea is to partition the memories up into fragments of data structures and then consider joining fragments individually. The valuation transformer Ψ gives the partition for each input memory and how they map to regions in the output. For fragments that are changing across iterations, we weaken them by trying to fold them into complete or partial checker edges. The central challenge is how to divide the memories up into fragments (essentially, deciding which regions should be weakened and which should not). Different partitions yield different upper bounds: ones with more precision in one region but less in another or vice versa. Thus, determining what partition yields the "best" upper bound *a priori* is very difficult. The novel aspect of our proposal is that we delay deciding as long as possible by incrementally matching fragments during the rewriting. This delayed partitioning is enabled by moving to binary join and widening operators. A way to view unary canonicalization operations (e.g., as in Sagiv et al. [2002] and Distefano et al. [2006]) is that they fix a method to decide on a partition before weakening.

I first summarize the fragment rewriting rules and give an intuition for their design. To see how the rules are applied, I present a number of examples later in this section (Example 4.7, Example 4.8, and Example 4.9). The first two rules (j-pt and j-chk) are *matching rules* that apply where the fragments on both sides correspond. The remaining rules are *weakening rules* where at least one side must be weakened in order to obtain an upper bound. Rules j-wchk and j-wseg apply when one side is already a checker edge (either complete or partial, respectively) thereby providing an indication on what to weaken to on the other side. Finally, rule j-waliases is particularly important, as it introduces a segment as a weakening for both sides. The weakening on the left (from emp) is justified by one of basic properties of inductive segments (Lemma 3.7). Observe that the weakening rules only apply once the partitioning has been determined. In this way, we use the matching rules to define a partitioning and guide where to weaken.

Lemma 4.12 (Soundness of fragment rewriting)**.** *The fragment rewriting rules preserve in-*

clusion in the concretizations.

If $\langle m_\ell, m_r\rangle \rightsquigarrow_{\Psi}^{F_\ell,F_r} m$*, then*

1. *if* $(\sigma_\ell,\nu_\ell) \in \boldsymbol{\gamma}(m_\ell)$ *and* $\nu_\ell \oslash \Psi_\ell$ *satisfies* F_ℓ*,*
   then $(\sigma_\ell, \nu_\ell \oslash \Psi_\ell) \in \boldsymbol{\gamma}(m)$*.*
2. *if* $(\sigma_r,\nu_r) \in \boldsymbol{\gamma}(m_r)$ *and* $\nu_r \oslash \Psi_r$ *satisfies* F_r*,*
   then $(\sigma_r, \nu_r \oslash \Psi_r) \in \boldsymbol{\gamma}(m)$*.*

Proof. By case analysis on the derivation of $\langle m_\ell, m_r\rangle \rightsquigarrow_{\Psi}^{F_\ell,F_r} m$. See Lemma A.6. □

Overall, the join in the shape domain is defined by iteratively rewriting fragments until we have consumed the inputs. The top-level rewriting rule is given in Definition 4.13. Intuitively, M is initially emp, and then we try to rewrite until M_1' and M_2' are emp in which case M' is the upper bound.

Definition 4.13 (Join in the shape domain)**.**

$$\boxed{(M_\ell \sqcup M_r)\ \circledast_{F_\ell,F_r}\ M\ \rightsquigarrow_\Psi\ (M_\ell' \sqcup M_r')\ \circledast_{F_\ell',F_r'}\ M'}$$

$$\frac{\langle m_\ell, m_r\rangle \rightsquigarrow_{\Psi}^{F_\ell',F_r'} m}{(M_\ell * m_\ell \sqcup M_r * m_r)\ \circledast_{F_\ell,F_r}\ M\ \rightsquigarrow_\Psi\ (M_\ell \sqcup M_r)\ \circledast_{F_\ell\wedge F_\ell',F_r\wedge F_r'}\ M * m} \tag{4.1a}$$

The join in the shape domain is then defined as follows:

$$M_\ell \sqcup_{\Psi}^{F_\ell,F_r} M_r = M \tag{4.1b}$$

$$\text{if}\quad (M_\ell \sqcup M_r)\ \circledast_{\mathsf{t},\mathsf{t}}\ \mathsf{emp}\ \rightsquigarrow_\Psi^*\ (\mathsf{emp} \sqcup \mathsf{emp})\ \circledast_{F_\ell,F_r}\ M$$

$$M_\ell \sqcup_{\Psi}^{F_\ell,F_r} M_r = M * \top \tag{4.1c}$$

$$\text{if}\quad (M_\ell \sqcup M_r)\ \circledast_{\mathsf{t},\mathsf{t}}\ \mathsf{emp}\ \rightsquigarrow_\Psi^*\ (M_\ell' \sqcup M_r')\ \circledast_{F_\ell,F_r}\ M \qquad \textit{(otherwise)}$$

where $\rightsquigarrow^*$ *is the transitive closure of* $\rightsquigarrow$*. Algorithmically, we rewrite using the above rule* (4.1a) *as much as possible. If the inputs can be completely consumed, then we return the result* (4.1b)*. Otherwise, if the rewriting gets stuck, then we summarize the remainder with* $\top$ *(i.e., a summary region that describes all stores and cannot be unfolded)* (4.1c)*. This weakening step results in an enormous loss in precision that we would like avoid but can be done if necessary.*

Theorem 4.14 (Soundness of join in the shape domain)**.** *The join operation in the shape domain is sound.*

If $(M_\ell \sqcup M_r) \circledast_{F_\ell,F_r} M \rightsquigarrow_\Psi (M'_\ell \sqcup M'_r) \circledast_{F'_\ell,F'_r} M'$*, then*

1. *if* $(\sigma_\ell, \nu_\ell) \in \boldsymbol{\gamma}(M_\ell)$, $(\sigma, \nu_\ell \otimes \Psi_\ell) \in \boldsymbol{\gamma}(M)$, $\nu_\ell \otimes \Psi_\ell$ *satisfies* F'_ℓ*, and* σ_ℓ, σ *are disjoint,*
   then $(\sigma'_\ell, \nu_\ell) \in \boldsymbol{\gamma}(M'_\ell)$, $(\sigma', \nu_\ell \otimes \Psi_\ell) \in \boldsymbol{\gamma}(M)$,
   $\sigma_\ell * \sigma = \sigma'_\ell * \sigma'$ *(for some* σ'_ℓ, σ'*).*
2. *if* $(\sigma_r, \nu_r) \in \boldsymbol{\gamma}(M_r)$, $(\sigma, \nu_r \otimes \Psi_r) \in \boldsymbol{\gamma}(M)$, $\nu_r \otimes \Psi_r$ *satisfies* F'_r*, and* σ_r, σ *are disjoint,*
   then $(\sigma'_r, \nu_r) \in \boldsymbol{\gamma}(M'_r)$, $(\sigma', \nu_r \otimes \Psi_r) \in \boldsymbol{\gamma}(M)$,
   $\sigma_r * \sigma = \sigma'_r * \sigma'$ *(for some* σ'_r, σ'*).*

As a consequence,

if $M_\ell \sqcup_\Psi^{F_\ell,F_r} M_r = M$*, then*

1. *if* $(\sigma_\ell, \nu_\ell) \in \boldsymbol{\gamma}(M_\ell)$ *and* $\nu_\ell \otimes \Psi_\ell$ *satisfies* F_ℓ*, then* $(\sigma_\ell, \nu_\ell \otimes \Psi_\ell) \in \boldsymbol{\gamma}(M)$.
2. *if* $(\sigma_r, \nu_r) \in \boldsymbol{\gamma}(M_r)$ *and* $\nu_r \otimes \Psi_r$ *satisfies* F_r*, then* $(\sigma_r, \nu_r \otimes \Psi_r) \in \boldsymbol{\gamma}(M)$.

Proof. By the soundness of fragment rewriting (Lemma 4.12). Observe that in the rewriting rule ((4.1a) in Definition 4.13) the input fragments are consumed and the output fragment is added to the result, which allows us to conclude that $\sigma_\ell * \sigma = \sigma'_\ell * \sigma'$ (or $\sigma_r * \sigma = \sigma'_r * \sigma'$).

The consequence is proven by induction on the derivation of the multistep rewriting and using Lemma A.7 as needed, which says that satisfaction of residual constraints is closed under backwards rewriting. □

Join in the Combined Domain

Like the comparison operator, the join operator for the combined domain first computes the join in the shape domain, discharges the residual conditions from unfolding, and then performs a join in the data domain. I write $\sqcup_{\mathbb{P}^\sharp}$ for the join operator in the data domain, which should satisfy the following condition:

$$\textit{If } (P_\ell \otimes \Psi_\ell) \sqcup_{\mathbb{P}^\sharp} (P_r \otimes \Psi_r) = P, \textit{ then } \boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P_\ell \otimes \Psi_\ell) \cup \boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P_r \otimes \Psi_r) \subseteq \boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P).$$

Definition 4.15 (Join in the combined shape and data domain)**.**

$$\begin{aligned}
&\langle M_\ell, P_\ell\rangle \sqcup_\Psi \langle M_r, P_r\rangle = \langle M, P\rangle \\
&\quad \textit{if} \quad M_\ell \sqcup_\Psi^{F_\ell, F_r} M_r = M \\
&\qquad\quad \textbf{prove}_{\mathbb{P}^\sharp}(P_\ell \otimes \Psi_\ell, F_\ell),\ \textbf{prove}_{\mathbb{P}^\sharp}(P_r \otimes \Psi_r, F_r),\ \textit{and} \\
&\qquad\quad (P_\ell \otimes \Psi_\ell) \sqcup_{\mathbb{P}^\sharp} (P_r \otimes \Psi_r) = P
\end{aligned}$$

Moreover, $\langle E_\ell, M_\ell, P_\ell\rangle \sqcup \langle E_r, M_r, P_r\rangle = \langle E, M, P\rangle$ *if the above join evaluates successfully when started with* $\Psi = \Psi_{\mathrm{init}}$ *and where* E *is the environment computed from* E_ℓ *and* E_r *during initialization.*

Theorem 4.16 (Soundness of join in the combined shape and data domain)**.** *The join operation is sound.*

If $\langle M_\ell, P_\ell\rangle \sqcup_\Psi \langle M_r, P_r\rangle = \langle M, P\rangle$, *then*

1. *if* $(\sigma_\ell, \nu_\ell) \in \langle M_\ell, P_\ell\rangle$, *then* $(\sigma_\ell, \nu_\ell \otimes \Psi_\ell) \in \langle M, P\rangle$.
2. *if* $(\sigma_r, \nu_r) \in \langle M_r, P_r\rangle$, *then* $(\sigma_r, \nu_r \otimes \Psi_r) \in \langle M, P\rangle$.

As a consequence,

$$\begin{aligned}&\textit{if}\ \langle E_\ell, M_\ell, P_\ell\rangle \sqcup \langle E_r, M_r, P_r\rangle = \langle E, M, P\rangle,\\ &\textit{then}\ \boldsymbol{\gamma}(\,\langle E_\ell, M_\ell, P_\ell\rangle\,) \cup \boldsymbol{\gamma}(\,\langle E_r, M_r, P_r\rangle\,) \subseteq \boldsymbol{\gamma}(\,\langle E, M, P\rangle\,).\end{aligned}$$

Proof. Direct by the soundness of join in the shape domain (Theorem 4.14) and the soundness conditions on $\boldsymbol{prove}_{\mathbb{P}^\sharp}$ and $\sqcup_{\mathbb{P}^\sharp}$ in the data domain. ☐

Widening

A widening operator $\triangledown$ is a join operator with a stabilizing property so as to ensure termination of the analysis [Cousot and Cousot 1977]. This operator should ensure that both shape and data invariants are stable after finitely many iterations. The shape join already has the stabilizing property, so a widening operation for the combined domain can be obtained by simply using the widening operator $\triangledown_{\mathbb{P}^\sharp}$ instead of the join operator $\sqcup_{\mathbb{P}^\sharp}$ in the data domain.

Theorem 4.17 (Stabilization of widening in the shape domain)**.** *Given any sequence of individual analysis states $\langle E'_n, M'_n, P'_n\rangle_{n\in\mathbb{N}}$, let $\langle E_n, M_n, P_n\rangle_{n\in\mathbb{N}}$ be the sequence defined as follows:*

$$\begin{aligned}\langle E_0, M_0, P_0\rangle &\stackrel{\text{def}}{=} \langle E'_0, M'_0, P'_0\rangle\\ \langle E_{n+1}, M_{n+1}, P_{n+1}\rangle &\stackrel{\text{def}}{=} \langle E_n, M_n, P_n\rangle \triangledown \langle E'_{n+1}, M'_{n+1}, P'_{n+1}\rangle\ .\end{aligned}$$

Then, the sequence $(M_n)_{n\in\mathbb{N}}$ (computed by joins in the shape domain defined in Definition 4.13) is ultimately stationary.

Proof. See Theorem A.8. ☐

Theorem 4.18 (Stabilization of widening in the combined shape and data domain). *The widening operator in the combined shape and data domain has the stabilizing property.*

Given any sequence of individual analysis states $\langle E'_n, M'_n, P'_n \rangle_{n \in \mathbb{N}}$, define the sequence of widenings $\langle E_n, M_n, P_n \rangle_{n \in \mathbb{N}}$ as in Theorem 4.17. Then, $\langle E_n, M_n, P_n \rangle_{n \in \mathbb{N}}$ is ultimately stationary.

Proof. Once the shape graphs stabilize as given by Theorem 4.17, the stabilizing property of the widening operator in the data domain $\triangledown_{\mathbb{P}^\sharp}$ ensures that the sequence of data elements $(P_n)_{n \in \mathbb{N}}$ converges. This argument is similar to those required to prove the stabilization of widening in cofibered domains [Venet 1996]. □

Examples

To get an idea how the rewriting rules in Figure 4.8 apply, I present a few examples that walk through the computation. In Example 4.7, we focus on computing the upper bound in the shape domain. Then, in Example 4.8 and Example 4.9, we look at how computing the join in the combined domain requires careful coordination between the shape and data domains.

Example 4.7 (Inferring a loop invariant for the skip list rebalancing code). Figure 4.9 shows a sequence of rewritings to compute the loop invariant in the skip list rebalancing example from Figure 2.5. The highlighting of nodes in the upper bound graph indicate the node pairings that are required to apply the rule, and the highlighting of edges in the input graphs show the fragments that are consumed in the rewriting step.

Line 1 shows the state after initialization: we have nodes in upper bound graph

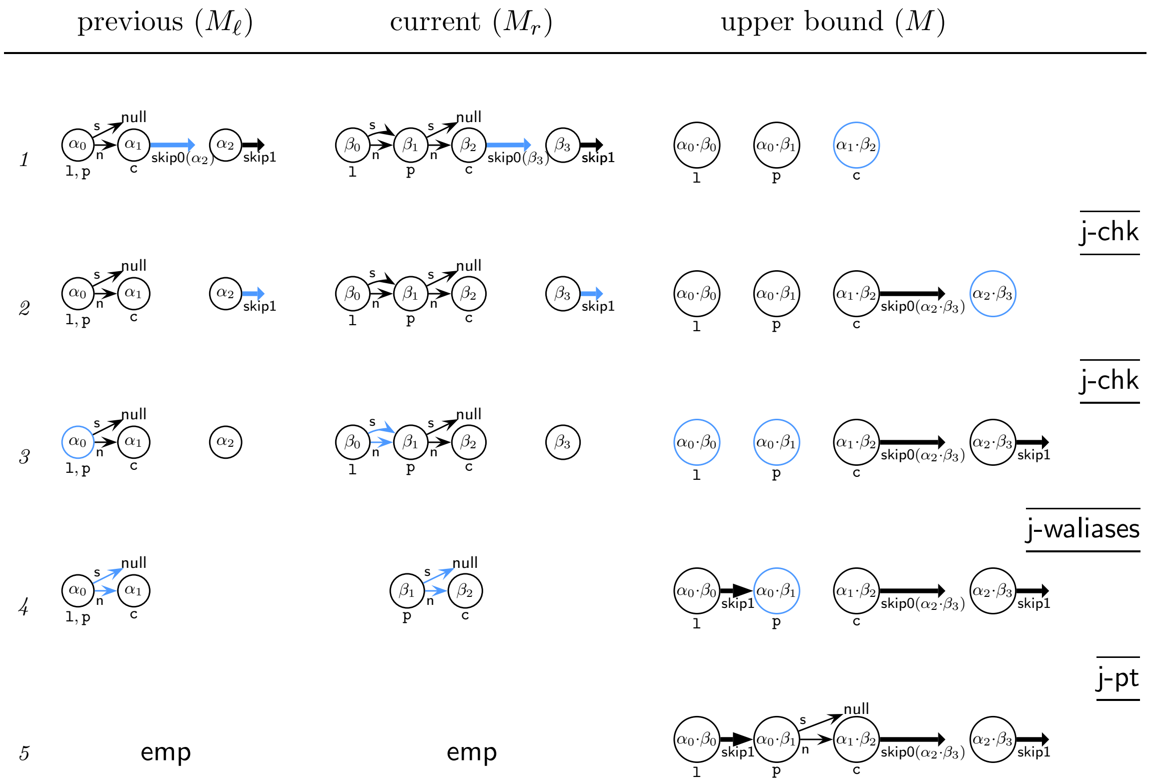
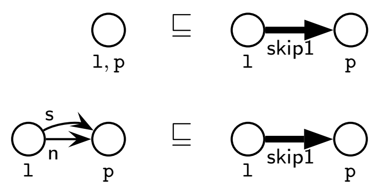


Figure 4.9: An example sequence of rewriting steps to compute an upper bound. The inputs are the graphs on the first iteration at program point 4 and point 8 in the skip list example from Figure 2.5. The fixed-point graph at point 4 is obtained by computing the upper bound of this result and the upper bound of the first-iteration graphs at program point 4 and program point 10. The valuation transformers Ψ_ℓ, Ψ_r are given implicitly by naming the nodes in the upper bound graph as pairs from the input graphs (i.e., a upper bound node γ is written as $\Psi_\ell(\gamma) \cdot \Psi_r(\gamma)$ in the above.) Also, the skip and next fields are abbreviated as s and n, respectively.

for the program variables. The first two steps (applying rule j-chk) match complete checker edges (first from $\alpha_1{\cdot}\beta_2$ and then from $\alpha_2{\cdot}\beta_3$). Note that the second application is enabled by the first where we add the mappings for $\alpha_2{\cdot}\beta_3$. Extra parameters are essentially implicit target nodes.

The core of the join is the three weakening rules where we fold memory regions. The next rule application j-waliases is a critical weakening step that introduces a segment (line 3 to 4). In this case, a node on one side corresponds to two nodes on the other ($\alpha_0{\cdot}\beta_0$ and $\alpha_0{\cdot}\beta_1$). This situation arises where on one side, we have must-alias information, while the other side does not (`l` and `p` are aliased on the left but not on the right). In this case, we weaken both sides to a segment. These weakenings are justified as follows:



(see Lemma 3.7)

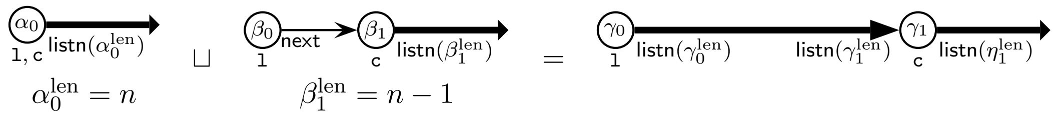
(see Example 4.5)

As shown on the top line, we weaken on the left by "splitting the node" and viewing it as an empty segment; this step is justified by Lemma 3.7. The j-waliases then checks that the segment is a weakening of the fragment on the right by using the comparison operation; the check that we need to perform here is the one shown in Example 4.5. Observe that we utilize the edge matching rules that populate Ψ to delineate the region to be folded (e.g., the region between β_0 and β_1 in the right graph). For the j-waliases rule, we do not specify here how the checker c is determined, but in practice, we can limit the checkers that need to be tried by, for example, tracking the type of the node (or looking at the fields used in outgoing points-to edges).

The last step is simply matching points-to edges. When we reach emp for M_ℓ and

M_r, then M is the upper bound.

Example 4.8 (Inferring a loop invariant for a list of given length traversal)**.** In this example, we consider the join of the first two iterates that arise during the traversal of a list of length n (checker listn from Example 3.3):



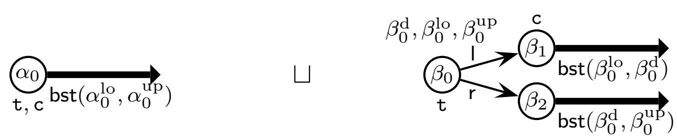
The join algorithm produces the shape invariant shown above by applying rules j-chk and j-waliases. Rule j-chk extends the valuation transformer so that $\Psi(\eta_1^{\text{len}}) = (\alpha_0^{\text{len}}, \beta_1^{\text{len}})$. From rule j-waliases, we get on the left side that $\gamma_0^{\text{len}} = \gamma_1^{\text{len}}$ (i.e., $\Psi_\ell(\gamma_0^{\text{len}}) = \Psi_\ell(\gamma_1^{\text{len}})$). On the right side, one unfolding step is required in the comparison (to fold from β_0 to β_1 into a listn segment), so by the definition of the additional parameter of the recursive call in the definition of checker listn, we have the relation that $\gamma_0^{\text{len}} = \gamma_1^{\text{len}} + 1$. This relation cannot be tracked by Ψ_r as I have defined it above (for presentation purposes), but we can consider an *extended valuation transformer* that not only maps nodes of the result into nodes of one of the inputs, but also allows expressing such relations among the nodes of the output graph. Such relations typically arise in the unfoldings performed during the comparisons required for applying rules j-waliases, j-wchk, and j-wseg.

After the join of the shape abstractions, the above relations are propagated to P_ℓ and P_r (i.e., by applying $\circledcirc$). This results in the following for the join in the data domain $\mathbb{P}^\sharp$:

$$\left[\eta_1^{\text{len}} = n \wedge \gamma_0^{\text{len}} = \gamma_1^{\text{len}}\right] \sqcup_{\mathbb{P}^\sharp} \left[\eta_1^{\text{len}} = n - 1 \wedge \gamma_0^{\text{len}} = \gamma_1^{\text{len}} + 1\right]$$

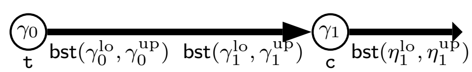
If we let $\mathbb{P}^\sharp$ be the domain of linear equalities Karr [1976], the result of the join is $[\gamma_0^{\text{len}} - \gamma_1^{\text{len}} = n - \eta_1^{\text{len}}]$, which says that the sum of the lengths corresponding to the partial and the complete checker segments is n (i.e., the length of the list does not change during the traversal). This invariant is the most precise one can hope for on this example.

Example 4.9 (Inferring a loop invariant for a search tree traversal)**.** Consider again analyzing the code for finding a value d in a binary search tree (i.e., basically, lines 2–8 in Figure 3.2). For this example, we assume $\mathbb{P}^\sharp$ is an abstract domain that supports inequalities among pairs of variables (e.g., octagons [Miné 2006]). Suppose in the first iteration, the cursor c is advanced to the left subtree (i.e., d is smaller than the data at the root), then the first widening is applied to the following arguments:



$$-\infty = \alpha_0^{\mathrm{lo}} < d < \alpha_0^{\mathrm{up}} = \infty \qquad -\infty = \beta_0^{\mathrm{lo}} < d < \beta_0^{\mathrm{d}} < \beta_0^{\mathrm{up}} = \infty \tag{4.2a}$$

The join in the shape domain yields the following shape graph:



(4.2b)

The valuation transformer Ψ is initialized to

$$\Psi(\gamma_0) = (\alpha_0, \beta_0) \quad \Psi(\gamma_1) = (\alpha_0, \beta_1)$$

from the environment. Then, rule j-chk applies to add the complete checker edge from γ_1, which also extends Ψ so that

$$\Psi(\eta_1^{\mathrm{lo}}) = (\alpha_0^{\mathrm{lo}}, \beta_0^{\mathrm{lo}}) \quad \Psi(\eta_1^{\mathrm{up}}) = (\alpha_0^{\mathrm{up}}, \beta_0^{\mathrm{d}}) \ .$$

Finally, rule j-waliases applies to create the segment from γ_0 to γ_1. This rule enriches Ψ so that the following relations hold:

$$\Psi_\ell(\gamma_0^{\mathrm{lo}}) = \Psi_\ell(\gamma_1^{\mathrm{lo}}) \qquad \Psi_\ell(\gamma_0^{\mathrm{up}}) = \Psi_\ell(\gamma_1^{\mathrm{up}}) \qquad \text{(for the initial state)}$$

$$\Psi_r(\gamma_0^{\mathrm{lo}}) = \Psi_r(\gamma_1^{\mathrm{lo}}) = \beta_0^{\mathrm{lo}} \quad \Psi_r(\gamma_0^{\mathrm{up}}) = \beta_0^{\mathrm{up}} \qquad \Psi_r(\gamma_1^{\mathrm{up}}) = \beta_0^{\mathrm{d}} \qquad \text{(for the first iterate)}$$

Note that the inclusion check that we need in order to weaken the subgraph from β_0 to β_1 to a partial checker edge is the comparison shown in Example 4.6. The extensions to Ψ_r can be read from there.

Then, applying the valuation transformers Ψ_ℓ, Ψ_r to the respective input elements, the data invariants to join are as follows:

$$\begin{aligned} &\left[\gamma_0^{\mathrm{lo}} = \gamma_1^{\mathrm{lo}} \;\wedge\; \gamma_0^{\mathrm{up}} = \gamma_1^{\mathrm{up}} \;\wedge\; -\infty = \eta_1^{\mathrm{lo}} < d < \eta_1^{\mathrm{up}} = \infty\right] \\ \sqcup_{\mathbb{P}^\sharp} \;\; &\left[-\infty = \gamma_0^{\mathrm{lo}} = \gamma_1^{\mathrm{lo}} = \eta_1^{\mathrm{lo}} < d < \gamma_1^{\mathrm{up}} = \eta_1^{\mathrm{up}} < \gamma_0^{\mathrm{up}} = \infty\right] \end{aligned}$$

However, the join of these two data invariants is problematic because the first invariant is, in a sense, too general, for any $(\gamma_0^{\mathrm{lo}}, \gamma_0^{\mathrm{up}}) = (\gamma_1^{\mathrm{lo}}, \gamma_1^{\mathrm{up}})$ approximates a 0-step segment. Specifically, the equality constraint $(\gamma_0^{\mathrm{lo}}, \gamma_0^{\mathrm{up}}) = (\eta_0^{\mathrm{lo}}, \eta_0^{\mathrm{up}})$ is not required in the initial state (i.e., the left data element) but is required in the first iterate (i.e., the right data element). Moreover, the segment between γ_0 and γ_1 is only an approximation of the corresponding subgraph on the right when $(\gamma_0^{\mathrm{lo}}, \gamma_0^{\mathrm{up}}) = (-\infty, \infty)$.

This example illustrates one of the difficulties that we sketched at the beginning of this chapter. As the symbolic values form the coordinates of the base data domain, it is quite sensitive to large changes in the shape graph. Here, we see that between the graph at the initial state and the join, α_0 has been "split" into γ_0 and γ_1 and $(\gamma_0^{\mathrm{lo}}, \gamma_0^{\mathrm{up}})$ has been "split" into $(\gamma_0^{\mathrm{lo}}, \gamma_0^{\mathrm{up}})$ and $(\gamma_1^{\mathrm{lo}}, \gamma_1^{\mathrm{up}})$. However, we observe that this becomes a non-issue once the graph stabilizes. Therefore, we propose to delay the join of the data invariants until the next iteration, which is a common static analysis technique.

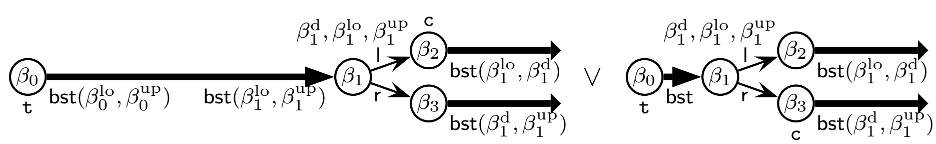
Now, for the case where the cursor is advanced to the right subtree in the first iteration, the result of the widening yields the same shape graph shown above in display (4.2b). The data constraints are, however, as follows:

$$-\infty = \gamma_0^{\mathrm{lo}} < \gamma_1^{\mathrm{lo}} = \eta_1^{\mathrm{lo}} < d < \gamma_1^{\mathrm{up}} = \eta_1^{\mathrm{up}} = \gamma_0^{\mathrm{up}} = \infty$$

The join of the numerical invariants corresponding to the left and right branches after one iteration is as follows:

$$-\infty = \gamma_0^{\mathrm{lo}} \leq \gamma_1^{\mathrm{lo}} = \eta_1^{\mathrm{lo}} < d < \gamma_1^{\mathrm{up}} = \eta_1^{\mathrm{up}} \leq \gamma_0^{\mathrm{up}} = \infty \tag{4.2c}$$

After the next iteration, we get the following two shape graphs:



The computation of the join of each of these invariants with the result of the first widening output (4.2b) reveals that the latter is stable at the shape level. Furthermore, from this point, the data invariant (4.2c) above is also stable, so we have obtained a fixed point. The loop invariant says that at any step of the find, the cursor `c` points to a subtree of `t` where the range of the data values in the subtree contains d.

Example 4.9 also shows the other difficulty alluded to at the beginning of this chapter, which was solved by a different technique. In the above, the range of the subtree is not only expressed in the checker parameters of a folded region but also as fields of unfolded nodes. Without these fields, the resulting situation of the first widening (shown in display (4.2a)) is similar to what is described above without the delayed join of data invariants. Specifically, we would get a "too general" instantiation of the partial checker edge where the lower bound at the head (γ_0^{lo}) could be any value smaller than the key at the root. The valuation transformer Ψ_r is never constrained so that $\Psi_r(\gamma_0^{\mathrm{lo}}) = \Psi_r(\gamma_1^{\mathrm{lo}}) = \beta_0^{\mathrm{lo}}$. This folding would be sound, but it would not allow folding at the next step, due to being too general. Instead, with these fields, we break the dependence on synthesizing the appropriate "less general" parameters. Example 4.8 does not require such fields because of the tight constraints on the parameters. Note that this kind of technique is also rather common in verification (e.g., McPeak and Necula [2005]), which we apply here to separate the analysis concerns from the modeling ones.

Implementation

In this subsection, we have described the join and widen algorithms through examples and with a focus on soundness. We now discuss some implementation concerns.

A Strategy for Applying Rewrite Rules. Unlike the comparison operation, the upper bound rules as described in Figure 4.8 have a fair amount of non-determinism, and unfortunately, applying the rules in different orders may yield different results in terms of precision. To avoid an exponential explosion in computational complexity, we fix a particular strategy in which to apply the rules, which has been determined, in part, experimentally. Note, however, that neither soundness nor termination is affected by the strategy that we choose. Intuitively, we obtain a good result when we are able to consume all the edges in the input graphs by applying the upper bound rules. A potential bad interaction between the rules is if we prematurely match (and consume) points-to edges that rather should be weakened together with other edges. For example, in Figure 4.9 (Example 4.7) before j-waliases (line 3), if instead we match the points-to edges $\alpha_0@\mathsf{n} \mapsto \alpha_1$ on the left and $\beta_0@\mathsf{n} \mapsto \beta_1$ on the right (i.e., apply j-pt) creating the node $\alpha_1{\cdot}\beta_1$, then we will not be able to consume all edges with further rewriting. Our strategy is to first exhaustively match complete checker edges (j-chk), as it does not prohibit any other rules and corresponds to identifying the "yet to be explored tail of the structure". Then, since the weakening rules for segments (j-waliases and j-wseg) only apply once we have identified corresponding regions (and that can only be consumed by performing this weakening), we apply these rules exhaustively when applicable. Then, to identify such regions, we apply j-pt but incrementally (i.e., we match one pair of points-to edges and restart). Finally, when nothing else applies, we try weakenings to complete checker edges (j-wchk).

Synthesis of Complete Checker Edges. The fragment rewriting rules for computing an upper bound in the shape domain (Figure 4.8) are based on using the iteration history to guide the weakening process. In particular, the hypothesis is that summaries in previous iterations give an indication where "unfolded regions" in the current iteration should be folded. This design principle translates directly to the weakening rules j-wchk and j-wseg. The j-waliases introduces a partial checker edge (i.e., segment), which is directed by the presence of aliasing in the previous iteration but not in the current iteration. However, there is no corresponding rule for introducing complete checker edges, which in part is based on our design principle. Such a rule would need to guess a folding with no history information, essentially requiring a generic canonicalization operation. Making a good guess becomes particularly difficult when there are additional parameters where we need to find appropriate instantiations and which may be involved in complex data constraints. From the user's perspective, making bad guesses would be particularly unintuitive, as the analyzer would have come up with an unintended data structure. Furthermore, such situations where the analyzer needs to synthesize a complete checker edge arise in relatively few places. They come up when a new structure is being constructed, which could be resolved by, for example, using constructor functions with postconditions asserting the appropriate checkers. In our implementation, we optionally associate checker definitions with type definitions. Then, when an object of a particular type is `malloc`ed, then we assert the associated checker invariant with that object.

Widening Disjunctive Analysis States. In general, we consider widening disjunctions of states

$$\langle E_0, M_0, P_0 \rangle \vee \cdots \vee \langle E_n, M_n, P_n \rangle \quad \triangledown \quad \langle E_0, M_0, P_0 \rangle \vee \cdots \vee \langle E_m, M_m, P_m \rangle \ .$$

The widening operator for disjunctions is based on the operator for individual states and attempts to find pairs of graphs that can be widened precisely in the sense that no region needs to be weakened to $\top$ (i.e., because an input region cannot be matched). In addition to this selective widening process, the widening may leave additional disjuncts, up to some fixed limit (perhaps based on trace partitioning [Rival and Mauborgne 2007]). More precisely, consider the following two disjunctions of states:

$$\langle M_0, P_0 \rangle \vee \cdots \vee \langle M_i, P_i \rangle \vee \cdots \vee \langle M_n, P_n \rangle \ \text{ and } \ \langle M_0', P_0' \rangle \vee \cdots \vee \langle M_j', P_j' \rangle \vee \cdots \vee \langle M_m', P_m \rangle$$

where we omit the environment for presentation purposes. Then, the widening on the two disjunctive states is described informally as follows:

- For each M'_j, if there exists an element M_i such that the rewriting for the upper bound in the shape domain for $M_i \sqcup M'_j$ does not get stuck (Definition 4.13), then we add $\langle M_i, P_i \rangle \triangledown \langle M'_j, P'_j \rangle$ to the result. If there exists no such element M_i, then we add $\langle M'_j, P'_j \rangle$ to the result.

- For each disjunct M_i such that no $\langle M_i, P_i \rangle \triangledown \langle M'_j, P'_j \rangle$ has been added to the result, then we add $\langle M_i, P_i \rangle$ to the result unless adding it would cause the generation of more disjuncts than a fixed constant. In this case, an $\langle M'_j, P'_j \rangle$ should be widened against $\langle M_i, P_i \rangle$ (with unmatched regions weakened to $\top$ if necessary).

Termination follows from the stabilizing property of the widening operator for individual analysis states (Theorem 4.18) and from the bound on the number of disjuncts.

Chapter 5

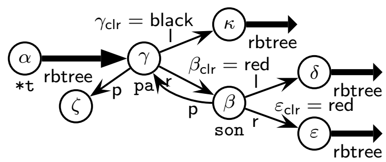
Discussion

To conclude the detailed discussion, I complete the presentation of the red-black tree insertion example from Section 5.1. I then provide an experimental evaluation of our algorithm (Section 5.2) and a discussion of our experience applying it (Section 5.3).

5.1 Example: Red-Black Trees

We return to the red-black tree insertion example from Figure 3.2 to discuss how the invariants in the rebalancing loop after an insertion can be obtained. In Figure 5.1, I present one of the rebalancing cases in detail with the fixed-point invariants shown at key points. At program point 21, I show a loop invariant for the rebalancing loop that is sufficient to show that after the loop, α (pointed to by `*t`) is a red-black tree according to checker `rbtree`. The shape portion of the loop invariant indicates that δ and ε are red-black trees (with certain parameters) but perhaps not locally around β. In the data portion, we have the ordering property on the data (shown at the bottom), which is obtained in the

```
while (pa != null) {
  if (pa->r && pa->r->r
      && pa->r->clr == RED && pa->r->r->clr == RED) {
    son = pa->r;
```



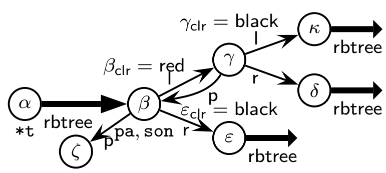
$$\kappa_{\mathrm{bh}} = \beta_{\mathrm{bh}} = \mathrm{ite}(\gamma_{\mathsf{clr}} = \mathsf{red}, \gamma_{\mathrm{bh}}, \gamma_{\mathrm{bh}} - 1)$$

$$\delta_{\mathrm{bh}} = \varepsilon_{\mathrm{bh}} = \mathrm{ite}(\beta_{\mathsf{clr}} = \mathsf{red}, \beta_{\mathrm{bh}}, \beta_{\mathrm{bh}} - 1)$$

$$\alpha_{\mathrm{bh}} - \beta'_{\mathrm{bh}} = bh - \beta_{\mathrm{bh}}$$

$$-\infty = \alpha_{\mathrm{lo}} \le \gamma_{\mathrm{lo}} \le \beta_{\mathrm{lo}} \le \varepsilon_{\mathrm{lo}} < \varepsilon_{\mathrm{up}} = \beta_{\mathsf{d}} = \delta_{\mathrm{lo}} < \delta_{\mathrm{up}} = \gamma_{\mathsf{d}} = \kappa_{\mathrm{lo}} < \kappa_{\mathrm{up}} \le \gamma_{\mathrm{up}} \le \alpha_{\mathrm{up}} = \infty$$

```
    son->r->clr = BLACK;
    son->p = pa->p;  set the l or r field of pa->p to replace pa with son;
    pa->r = son->l;  if (son->l) { son->l->p = pa; }
    son->l = pa;     pa->p = son;
    pa = son;
```



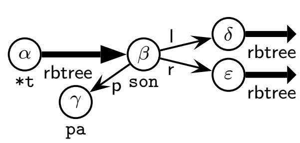
$$\kappa_{\mathrm{bh}} = \beta_{\mathrm{bh}} = \mathrm{ite}(\gamma_{\mathsf{clr}} = \mathsf{red}, \gamma_{\mathrm{bh}}, \gamma_{\mathrm{bh}} - 1)$$

$$\delta_{\mathrm{bh}} = \varepsilon_{\mathrm{bh}} = \mathrm{ite}(\beta_{\mathsf{clr}} = \mathsf{red}, \beta_{\mathrm{bh}}, \beta_{\mathrm{bh}} - 1)$$

$$\alpha_{\mathrm{bh}} - \beta'_{\mathrm{bh}} = bh - \beta_{\mathrm{bh}}$$

$$-\infty = \alpha_{\mathrm{lo}} \le \gamma_{\mathrm{lo}} \le \beta_{\mathrm{lo}} \le \varepsilon_{\mathrm{lo}} < \varepsilon_{\mathrm{up}} = \beta_{\mathsf{d}} = \delta_{\mathrm{lo}} < \delta_{\mathrm{up}} = \gamma_{\mathsf{d}} = \kappa_{\mathrm{lo}} < \kappa_{\mathrm{up}} \le \gamma_{\mathrm{up}} \le \alpha_{\mathrm{up}} = \infty$$

```
  }
  ... other rebalancing cases ...
  son = pa; pa = pa->p;
```



$$\delta_{\mathrm{bh}} = \varepsilon_{\mathrm{bh}} = \mathrm{ite}(\beta_{\mathsf{clr}} = \mathsf{red}, \beta_{\mathrm{bh}}, \beta_{\mathrm{bh}} - 1)$$

$$\alpha_{\mathrm{bh}} - \beta'_{\mathrm{bh}} = bh - \beta_{\mathrm{bh}}$$

$$-\infty = \alpha_{\mathrm{lo}} \le \beta_{\mathrm{lo}} \le \varepsilon_{\mathrm{lo}} < \varepsilon_{\mathrm{up}} = \beta_{\mathsf{d}} = \delta_{\mathrm{lo}} < \delta_{\mathrm{up}} \le \beta_{\mathrm{up}} \le \alpha_{\mathrm{up}} = \infty$$

```
}
```

at fixed point

Figure 5.1: Rebalancing in the red-black tree insertion example from Figure 3.2.

search loop (lines 2–8 in Figure 3.2) prior to the insertion. Note that this ordering invariant is obtained by the analysis algorithm as described in the example widening on a binary search tree traversal (Example 4.9 of Section 4.2.2) and then preserved in this loop. The other data constraints describe the invariant on the black height parameters of the checkers (e.g., β_{bh}). The top constraint gives the relation between the black height at β with those at δ and ε, which first comes from the unfolding of the rbtree in the search loop and then notably preserved on insertion (line 9 in Figure 3.2). The middle constraint is the relation between the black height at α with the subtree at β (where bh is the initial black height of the entire tree and β'_{bh} is the black height checker parameter at the end of segment to β—as opposed to β_{bh}, which is the black height parameter from β). This invariant is also obtained in the search loop and in the same manner as the example widening on lists of given length (Example 4.8 of Section 4.2.2). However, the base domain should be richer to handle the additional Boolean structure (on whether a node is red or black) by using, for example, binary decision diagrams (BDDs) with linear equalities at the leaves.

Observe that we have no constraints on the red-ok parameters (e.g., $\varepsilon_{\mathrm{redok}}$) meaning that any of δ, ε, and β may be red and thus locally violating the color aspect of the red-black tree invariant for the entire structure. The shown rebalancing case addresses this violation by performing a left rotation and coloring. From program point 21 to point 13, the analysis does a backward unfolding to materialize the fields of pa. This unfolding (along with $\alpha_{\mathrm{bh}} - \beta'_{\mathrm{bh}} = bh - \beta_{\mathrm{bh}}$) yields the additional black height constraint on κ_{bh} and β_{bh}. Then the condition that $\beta_{\mathsf{clr}} = \mathsf{red}$ (and an unfolding constraint on $\varepsilon_{\mathrm{redok}}$) tells us that $\gamma_{\mathsf{clr}} = \mathsf{black}$. For compactness, I do not show the unfolding of ε, which is needed only to access its color field. Aside from the coloring of ε, the rotation only affects the graph (as shown at point 18). Now, compare this after-rotation state with the loop invariant at program point 21. We see that the after-rotation state is contained in the loop invariant (after advancing the cursor `pa`) by folding the region from γ into a `rbtree`, which is computed by the join as described in Section 4.2. In the data constraints, the key observation is that the coloring gives us that $\kappa_{\mathrm{bh}} = \delta_{\mathrm{bh}} = \varepsilon_{\mathrm{bh}}$. Also, while the new black height at β and ε increase by one, this is summarized by the difference equality constraint.

5.2 Experimental Evaluation

We evaluate our shape analysis using an implementation for analyzing C code. Our analysis is written in OCaml and uses the CIL infrastructure [Necula et al. 2002]. We have applied our analysis to a number of data structure manipulation benchmarks and a larger Linux device driver benchmark (`scull`). Table 5.1 presents analysis statistics executed on a 2.0 GHz Intel Xeon with 2 GB of RAM where each timing result is the mean time over 10 runs. In each case, we verified that the pointer manipulation preserved the shape invariants of the data structures (e.g., back-pointer property, acyclicity, non-sharing, treeness) as given by a checker. When the operation exists for the both the back and non-back pointer version of the data structure (e.g., doubly-linked list versus singly-linked list), we present analysis statistics for both as a point of comparison. The “doubly-linked list remove, single cursor” benchmark is a variant where the search for the element to remove is done with only one cursor, so back pointers are required to perform the operation. The “doubly-linked list remove and loop back” example finds an element if it exists, removes it, and walks back

Benchmark	Analysis Time (ms)	Max. Disj. (num)	Max. Iter. (num)	Widening Time (ms)	Widening Time (fraction)
singly-linked list reverse	1.0	1	3	0.5	0.50
doubly-linked list reverse	1.5	1	3	1.1	0.74
singly-linked list create	1.2	1	3	0.9	0.72
doubly-linked list create	3.0	1	3	2.5	0.82
singly-linked list find	3.0	1	4	2.7	0.89
doubly-linked list find	3.6	1	4	3.0	0.84
singly-linked list copy	4.4	2	3	3.8	0.86
doubly-linked list copy	5.4	2	3	4.7	0.87
singly-linked list insert	8.5	2	4	7.6	0.89
doubly-linked list insert	5.9	2	4	5.0	0.86
singly-linked list remove	8.6	2	4	7.6	0.88
doubly-linked list remove	10.4	3	4	9.4	0.91
doubly-linked list remove, single cursor	17.9	5	4	16.8	0.94
doubly-linked list find and loop back	5.9	1	4	5.2	0.87
doubly-linked list remove and loop back	18.1	5	4	16.8	0.93
search tree find	5.4	2	5	4.7	0.87
search tree with parent find	8.1	2	5	7.4	0.92
search tree insert	24.7	3	5	22.6	0.92
search tree with parent insert	16.6	3	5	14.7	0.89
search tree with parent find and loop back	15.3	2	5	14.2	0.93
search tree with parent insert and loop back	64.7	5	5	62.3	0.96
two-level skip list `rebalance`	11.7	1	7	10.7	0.92
Linux `scull` driver (894 lines of C code)	3,969.6	4	16	3,895.3	0.98

Table 5.1: Benchmark results for verifying shape preservation. The columns show the total analysis time in milliseconds, the maximum number of disjuncts at any program point, the maximum number of iterations at any point, the total time spent in widening, and the fraction of time spent in widening.

modifying the previous nodes (e.g., updating a length field); the other "loop back" examples are similar. We did not verify any numerical properties on the node data (e.g., ordering), as we do not yet have an effective interface to implementations of numerical base domains.

The first aspect we consider is the *expressiveness* of our algorithm. Section 3.2.1 discusses the kinds of shapes expressible as checkers, and here, we provide some experimental evidence by considering a wide range of data structures (e.g., lists or trees, with or without a back pointer, and even a skip list shape). While there are shapes beyond the reach of our current algorithm, our end-user shape analyzer enables the user to analyze custom structures that can be variants or combinations of these basic forms, as well as focus the analysis to the properties of interest to the user. This design choice is in contrast to the specialized analyzers that build in the shapes of interest (e.g., Berdine et al. [2007] focus on singly- and doubly-linked lists).

In the data structure manipulation benchmarks, the analysis times are mostly negligible, but more importantly, the maximum number of disjuncts (i.e., the number of shape graphs), we need to keep at any program point seems to be small. Keeping the number of disjuncts seems to be critical for getting good *efficiency*. For most of the operations that exist for both the back and non-back pointer versions of the data structure, the tracking and checking of the additional pointer adds a bit to the analysis time, but there are a few seeming anomalies. The non-back pointer version of insert for both lists and trees take more time than on the data structure with back pointers while the maximum number of disjuncts and iterations are the same. It turns out this difference is still due to the number of disjuncts. The additional information provided by the back pointer allows disjuncts to be collapsed at additional program points. For lists, the doubly-linked list insert has two disjuncts in one fewer program point (two to one), and for trees, the search tree with parent insert has three disjuncts in one fewer place (two to one) and two disjuncts in six fewer places (six to zero). While it is difficult to make timing comparisons with other shape analyses due to different input languages and properties analyzed, as well as the relatively small size of such benchmarks, we observe that shape analysis times for such data structure benchmarks are often measured in the deciseconds to seconds range. Bogudlov et al. [2007] report rather efficient analysis times for the generic TVLA analyzer, but there is still a cost to complete generality. As a point in comparison, they report the analysis times of 290 ms for singly-linked list reversal and 850 ms for insertion into a search trees where for each case, they verify that the data structure invariant is maintained (like in Table 5.1).

We also look at a larger benchmark in a Linux device driver. The `scull` driver is from the Linux 2.4 kernel and was used by McPeak and Necula [2005]. The main data structure used by the driver is an array of doubly-linked lists. Because we do not yet have support for arrays, we rewrote the array operations as linked-list operations (and ignored other `char` arrays). We analyzed each function individually by providing appropriate preconditions. All function calls were inlined, as our implementation does not yet support proper interprocedural analysis (but which adds an exponential cost in the analysis time). There has been prior work on interprocedural shape analysis that is likely to be applicable to our algorithm [Rinetzky et al. 2005; Gotsman et al. 2006]. One function (`cleanup_module`) was not completely analyzed because of an incomplete handling of the array issues; it is not included in the line count. Also, as discussed in Section 4.2, because we only fold into checkers edges based only on history information, sometimes we cannot generate the appropriate checker edge when a new structure is being constructed out of already existing cells. Specifically, the small bit of information we sometimes need is that this **null** pointer should be treated as, for example, the empty list. For these experiments, we use an annotation that adds a checker edge for this purpose. Such an annotation is used six times in the `scull` driver example (and no times in all of the small data structure manipulation benchmarks). Observe that in this larger example, the number of disjuncts we need to maintain at any program point (i.e., the number of graphs) seems to stay reasonably low.

In the two rightmost columns of Table 5.1, I show the portion of the analysis time that is spent in the widening operation. It is the dominant factor in the analysis time. Unlike updating the abstract memory state, our widening algorithm as described in Section 4.2 is a global operation over the memory graphs. Each call to widening requires a traversal over the input graphs from the program variables to identify the regions of memory that are similar. In general, we expect that across iterations of a loop, there are portions of memory that simply have not been touched by the body of the loop. Ideally, the widening operation should also be able to take advantage of the disjointness constraint on memory regions so that these portions do not need to be traversed.

5.3 Experience Discussion

In this section, we discuss our experience applying our analyzer, as well as current limitations and possible extensions of our algorithm.

Writing Data Structure Invariant Checkers for Shape Analysis

Our experience with providing checkers for analysis has been that they are quite easy to write because they describe only the steady-state invariant of the data structure and only need to be defined on a per-program basis. The disjointness restriction from separation logic also makes the specification quite compact for data structures that make only careful use of sharing (e.g., red-black trees in a few lines (Figure 3.1(b))). Like most specifications, it is more difficult to "reverse engineer" the appropriate invariant checker for someone else's code (as we did for the `scull` driver example). We view this observation as another reason for end-user program analysis and making specifications accessible to the developer.

There are aspects where usability can still be improved. We have found that sometimes one would like to define data structure checkers polymorphically, that is, leave the element kind unspecified or given by a parameter (either in terms of the shape or the data properties of each node). This kind of extension has been described by Berdine et al. [2007] for doubly-linked lists. They define a higher-order list segment predicate that is parameterized by the shape of the "node" to add a level of polymorphism. In contrast, we can describe custom structures monomorphically with the appropriate checker definitions. Also, it might be convenient to automatically generate default checker definitions from type definitions to cover simple cases. For example, the generated definitions could be tree-like checkers that assume no sharing and thus only require the user to override the defaults when the data structures of interest are more complex.

Static analysis and verification in general is always with respect to a specification of what it means for the code to be correct, which leads to the issue of making sure that the specifications convey to the analysis what is intended by the user. For our end-user shape analysis, the checker definitions are specifications that we take as axioms. Users are responsible for ensuring that the checker definitions they provide correspond to the data structure invariants they have in mind. For example, it is possible that a checker definition describes no concrete stores. At the same time, here is a situation where it is particularly helpful that view checkers as executable code. With an appropriate compilation of checker definitions, the user can utilize run-time testing to refine the specifications and ensure they correspond to their intent. In this scenario, testing complements static analysis. Specifically, on runs of the simplest, expected test cases, if an invariant checker fails, then it is likely that there is a disconnect between the specification and the developer's intent. The specification can then be quickly fixed. In turn, once there is reasonable confidence in the checker definitions, our static shape analysis can assist in obtaining full coverage of the program code (i.e., covering the unexpected corner cases). This concept that the same specification should be used in both static checking and run-time checking does appear elsewhere, such as the Spec$^{\#}$ system [Barnett et al. 2004] where first-order logic specifications are compiled into run-time checks.

Limitations and Possible Extensions

Section 3.2 discusses the class of data structure invariant checkers that we consider in this work. Generally speaking, they capture invariants of a single structure where any global property of the structure is broken down into a sequence of local checks. This class does not include global properties between structures, such as, the invariant that two lists have the same set of elements. Thinking in terms of run-time invariant checking code, such specifications might be implemented by manipulating global state or using an auxiliary container (e.g., a set-valued checker parameter), but both methods are outside the class of checkers we consider.

Our end-user shape analysis algorithm consists of a number of techniques to materialize and summarize with checker-based summaries. While they are designed to be applicable in many data structure manipulation contexts, they are situations where other kinds of summaries may be useful. For example, we could have partial checker edges with multiple holes that would allow us to summarize memory regions with more than two endpoints. The segments defined in this chapter (Section 3.3.1) work well for traversals that use cursors along a path through the structure, but perhaps not as well for code that uses multiple cursors along different branches of a structure.

In order to perform precise analysis on a larger class of programs, the clearest hurdles are to cope with other ways C programs manipulate memory, specifically arrays and pointer arithmetic. As mentioned already, we do not yet have any integration of array reasoning techniques in our implementation. To reason precisely about the contents of arrays, notions of materialization and summaries are also useful, so it might be fruitful to consider extending our approach to reason about them. In some ways arrays are more constrained than pointer-based data structures, as it is one fixed shape and disjointness of array cells is a given. On the contrary, the access patterns for arrays are much more varied. They can be treated like unidirectional containers (cf., singly-linked lists) or bidirectional ones (cf., doubly-linked lists), traversed like search trees in divide-and-conquer routines, used as binary heaps, or accessed seemingly at random (e.g., in hash tables). In the context of end-user program analysis, we would like to find ways in which the user can help the analysis come up with the appropriate summaries when the uses can be so varied.

The only kind of addressing expression we currently consider is field offset (i.e., $\alpha@f$). To perform shape analysis on code with pointer arithmetic, we need to extend the kinds of addressing expressions. The main challenge with such an extension is for any addressing expression, we need to determine which point-to edge it correspond to if it exists (i.e., decide equality amongst addressing expressions) or otherwise perform the appropriate unfolding to materialize it. The disjointness constraint and the framework of separation logic does allows us to avoid proving disequality constraints amongst addressing expressions while still performing destructive updates of the analysis state.

Chapter 6

Conclusions and Future Work

The demand for program analysis tools will likely continue to rise, as we look to any technique that can help us deal with the ever-growing complexity and importance of software systems. Yet, if program analyzers continue to be viewed as expert tools, it seems that a broad adoption of such tools into software engineering processes will be very difficult.

I have argued that because of the fundamental limitations in program analysis, there have been two conflicting forces in program analysis design. On one hand, we are driven towards *generic* verification frameworks that apply broadly with respect to both programs and properties but expect a high-level of program analysis expertise from users. On the other hand, *specialized* analyzers that expect no analysis expertise but also allow no input from users (i.e., have built-in domain-specific knowledge tailored to particular classes of users). In contrast, my thesis advocates a slight shift in mentality where we look to users to cooperate with the analyzer but without expecting users to be program analysis experts. In other words, this dissertation makes a step towards realizing *end-user program* *analysis*—where a non-expert can interact with an analyzer to provide the domain-specific knowledge it needs in order to verify the property of interest to the user.

6.1 Summary of Contributions

The target user for most program analyzers are software developers, as they can use analysis tools to help them better understand and eliminate errors from their code. They are also the ones who have the domain-specific knowledge that can really boost an analysis tool. In this context, I have described a new shape analysis technique based on the end-user approach. Because of the high precision requirements for effective shape analysis, it is a particularly attractive case study for the end-user approach. In summary, this work makes the following contributions:

1. *Invariant checking code as specification for analysis.* We observe that data structure invariant checking code can help guide a shape analysis and provides a familiar mechanism for the developer to supply information to the analyzer. For the analyzer, invariant checkers describe the program-specific data structures of interest to the developer along with a hint on how they are used.

2. *End-user shape abstractions.* We develop an analysis abstraction that is built from the user-supplied invariant checkers. In this way, we involve the user in providing the fundamental elements for a good abstraction.

3. *Generalize from the end-user abstractions.* We introduce a generalization of invariant checkers to describe the memory state when the data structure invariant holds only

partially. This step is critical in bridging the gap between using invariant checking code for run-time testing and building an analysis abstraction out of it.

4. *Accommodate varying user specification.* We make our analysis more robust with respect to user specification by deriving additional information through a separate type analysis on the checker definitions. This phase augments the testing specifications with additional information necessary for static analysis.

5. *Local shape invariant inference.* We notice that the iteration history of the analysis can be used to guide the weakening of abstract memory states built out of data structure invariant checkers. Based on this observation, we design a *widening* operator so that the user is involved only in specification at the more intuitive global level.

6.2 Future Work

The more technical limitations and possible extensions of this work are discussed in Section 5.3. In this section, I focus on broader directions for extending the ideas described in this dissertation.

In this dissertation, we examine how to obtain specifications from the user in a way that is less obtrusive. Certainly, one line of interesting work is to broaden the class of invariant checkers that can be utilized or find other kinds of less obtrusive specifications. Yet, despite our best efforts, it seems likely that there will be facts about a program that are both beyond the limitations of what can be efficiently computed automatically and what we would hope to get from a non-expert user in terms of direct specification. In the following examples, the underlying theme is then to broaden the context in which the developer and the analyzer cooperate.

User Mediation between Run-Time and Static Checking

As previewed in Section 5.3, run-time checking (e.g., testing) and static checking (e.g., static analysis) have complementary strengths. Testing is more accessible and broadly applicable, while static analysis provides more complete coverage, yet it is rare that both techniques are consistently applied to the same software. However, if efforts applying one technique can improve the results in the other and vice versa, then there will be a real incentive to use both.

For example, based on our shape analysis with invariant checkers, one promising direction is to trade off complexity in static analysis for run-time execution of validation code (i.e., some form of hybrid checking). In Chapter 4, we assume that the base data domain is as expressive as needed to capture the data properties of interest, but in practice, we may not want incur the analysis overhead of a complex data domain (e.g., the one necessary to verify the red-black tree example of Section 5.1). The challenge is that it is not clear how to discharge some (but not all) of the complexity to run-time checks. Yet, with user-supplied data structure invariant checkers, the user provides some guidance because they indicate not only summarization of shape but also of data. For complete static checking as in this dissertation, upon unfolding a checker, the data constraints can be assumed, and upon folding, they must be proven. Since the user limits the kinds of summaries of interest, it may be clear what the folding must be from the shape alone and so any unprovable data constraints can be discharged using run-time checks. To sum up, while the general concept of hybrid checking is not new in itself, the unique aspect here is that it might be possible to involve the user in mediating a desirable trade-off.

Furthermore, from the user's perspective, the above scenario can be viewed as a way to minimize the run-time cost of instrumenting software with validation code. A data domain that can simply remember the data constraints from unfolding may be sufficient to prove folding conditions on unchanged parts of the structure, leaving only more complicated modifications to run-time checks. With such a framework, there would be a strong incentive to write validation code because not only can the same annotation be used for both static analysis and testing, but the work put into applying the static analysis would improve the run time of tests.

Development-Time Interaction

Because our shape analysis uses abstractions that reflect developer intent, it seems particularly amenable for a program understanding tool that shows how memory is manipulated by the code (i.e., a heap-aware symbolic debugger). I envision this tool would assist with the mental simulation of the program that the developer does when reasoning about a program fragment. It would allow the developer to think at a higher-level of abstraction by producing graphical diagrams analogous to those that would appear on a whiteboard while leaving the analysis to keep track of the low-level details.

Possible user interactions during development time in an integrated development environment (IDE) setting seem particularly interesting. In this setting, the user could restrict the shape analysis to apply only to the program fragments currently in development. For a precise analysis like ours, it becomes especially important to apply the analysis modularly and incrementally on small fragments. More concretely, the developer could instrument the code with calls to invariant checkers when it should hold (i.e., the data structure is in the steady state). The shape analysis could be applied only to the program fragment between the steady states where the developer is currently working and show the developer how the abstract memory state changes as modifications are made to the code and whether the postcondition for the next steady state can be proven. There are number of technical challenges, such as making sure the analysis can run incrementally and in interactive time. Nonetheless, by automating the mental simulation done by developers, there would be another incentive for them to write validation code.

Bibliography

Gilad Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *Static Analysis Symposium (SAS)*, pages 204–220, 2006.

Gilad Arnold, Roman Manevich, Mooly Sagiv, and Ran Shaham. Combining shape analyses by intersecting abstractions. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 33–48, 2006.

Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis of single-parent heaps. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 91–105, 2007.

Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking Software*, pages 103–122, 2001.

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods (IFM)*, pages 1–20, 2004.

Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram

Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

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

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

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

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

Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *Conference on Computer-Aided Verification (CAV)*, pages 221–225, 2007.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Static Analysis Symposium (SAS)*, pages 182–203, 2006.

Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 147–163, 2005.

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

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

David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310, 1990.

Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A reachability predicate for analyzing low-level software. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.

Sigmund Cherem and Radu Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, pages 52–71, 1981.

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

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

David Delmas and Jean Souyris. Astrée: From research to industry. In *Static Analysis Symposium (SAS)*, pages 437–451, 2007.

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

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

Joshua Dunfield. Refined typechecking with Stardust. In *Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 21–32, 2007.

Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, 2002.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17 (2), 2008.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science: Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

Pascal Fradet and Daniel Le Métayer. Shape types. In *Symposium on Principles of Programming Languages (POPL)*, pages 27–39, 1997.

Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–15, 1996.

Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *Symposium on Principles of Programming Languages (POPL)*, pages 338–350, 2005.

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

Sumit Gulwani and Ashish Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *Conference on Computer-Aided Verification (CAV)*, pages 379–392, 2007.

Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Symposium on Principles of Programming Languages (POPL)*, pages 235–246, 2008.

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

Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Symposium on Principles of Programming Languages (POPL)*, pages 310–323, 2005.

Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 339–348, 2008.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, 1969.

Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.

Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

Gary A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symposium on Programming (ESOP)*, pages 124–140, 2005.

Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 26–38, 2000.

Tal Lev-Ami, Neil Immerman, and Mooly Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Conference on Computer-Aided Verification (CAV)*, pages 547–561, 2006.

Nancy G. Leveson and Clark S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In *Conference on Computer-Aided Verification (CAV)*, pages 519–533, 2005.

Alexey Loginov, Thomas Reps, and Mooly Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Static Analysis Symposium (SAS)*, pages 261–279, 2006.

Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *Static Analysis Symposium (SAS)*, pages 419–436, 2007.

Roman Manevich, G. Ramalingam, John Field, Deepak Goyal, and Mooly Sagiv. Compactly representing first-order structures for static analysis. In *Static Analysis Symposium (SAS)*, pages 196–212, 2002.

Roman Manevich, Mooly Sagiv, G. Ramalingam, and John Field. Partially disjunctive heap abstraction. In *Static Analysis Symposium (SAS)*, pages 265–279, 2004.

Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Conference on Computer-Aided Verification (CAV)*, pages 476–490, 2005.

Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17 (3):348–375, 1978.

Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19 (1):31–100, 2006.

Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 221–231, 2001.

National Institute of Standards and Technology. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10, June 2002.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction (CC)*, pages 213–228, 2002.

Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In Byron Cook and Andreas Podelski, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2007. ISBN 978-3-540-69735-0.

Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 203–217, 2008.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*, chapter 2.6, pages 104–128. Springer, 1999.

William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.

G. Ramalingam, Alex Varshavsky, John Field, Deepak Goyal, and Mooly Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2002.

Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming (ESOP)*, pages 380–398, 2003.

Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Shape analysis and applications. In Y. N. Srikant and Priti Shankar, editors, *Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter 12. CRC Press, second edition, 2007.

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

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

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

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

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

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

Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin C. Rinard. Field constraint analysis. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 157–173, 2006.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Symposium on Principles of Programming Languages (POPL)*, pages 214–227, 1999.

Eran Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 25–34, 2004.

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Conference on Computer-Aided Verification (CAV)*, pages 385–398, 2008.

Appendix A

Proof Listings

This chapter provides more detailed proofs of some of the lemmas and theorems in this dissertation. I also include here some auxiliary lemmas that do not appear in the main text. When the lemmas or theorems are restatements of ones in the main text, I indicate it by providing the appropriate references. The chapter is organized around the different parts of our end-user abstraction and analyzer.

A.1 Checker Evaluation and Memory Abstraction

Theorem A.1 (Successful evaluations correspond to abstract memory states)**.**

Restatement of Theorem 3.3.

> *If* $\nu \vdash \langle \sigma, e \rangle \Downarrow \langle \sigma', \mathsf{true} \rangle$*, then* $\sigma = \sigma'' * \sigma'$ *and* $\langle \sigma'', \nu' \rangle \models \ulcorner\!\!\rightarrow(e)$ *(for some* $\nu' \supseteq \nu$ *and some* σ''*).*

In particular, if $\nu \vdash \langle \sigma, e \rangle \Downarrow \langle [\cdot], \mathsf{true} \rangle$*, then* $\langle \sigma, \nu' \rangle \models \ulcorner\!\!\rightarrow(e)$ *(for some* $\nu' \supseteq \nu$*).*

Proof. By induction on the given derivation.

Case:
$$\frac{\nu, \beta \to u \vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', \mathsf{true}\rangle}{\nu \vdash \langle [\nu(\alpha) + f \mapsto u] * \sigma, \mathsf{let}\ \beta = \alpha\,.\,f\ \mathsf{in}\ e\rangle \Downarrow \langle \sigma', \mathsf{true}\rangle}\ \textsf{e-read}$$

$\sigma = \sigma'' * \sigma'$ (for some σ'') — By i.h.

$\langle \sigma'', \nu'\rangle \models \Gamma(e)$ (for some $\nu' \supseteq \nu, \beta \to u$) — By i.h.

$\Gamma(e) = \langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle$ — By def. of $\Gamma(\cdot)$

$\langle \sigma'', \nu'\rangle \models \langle M_i, F_i\rangle$ (for some $i \in 0..n$) — By def. of $\models$

$\langle [\nu'(\alpha) + f \mapsto u] * \sigma'', \nu'\rangle \models \langle \alpha @ f \mapsto \beta * M_i, F_i\rangle$ — By def. of $\models$

$\langle [\nu'(\alpha) + f \mapsto u] * \sigma'', \nu'\rangle \models \Gamma(\mathsf{let}\ \beta = \alpha\,.\,f\ \mathsf{in}\ e)$ — By def. of $\models$ and $\Gamma(\cdot)$

$[\nu'(\alpha) + f \mapsto u] * \sigma = [\nu'(\alpha) + f \mapsto u] * \sigma'' * \sigma'$

Case:
$$\frac{\nu \vdash \langle \sigma, [\alpha, \delta/\pi, \rho]e\rangle \Downarrow \langle \sigma', \mathsf{true}\rangle \quad (\pi\,.\,c(\rho)\ := e)}{\nu \vdash \langle \sigma, \alpha\,.\,c(\delta)\rangle \Downarrow \langle \sigma', \mathsf{true}\rangle}\ \textsf{e-call}$$

$\sigma = \sigma'' * \sigma'$ (for some σ'') — By i.h.

$\langle \sigma'', \nu'\rangle \models \Gamma([\alpha, \delta/\pi, \rho]e)$ (for some $\nu' \supseteq \nu$) — By i.h.

$\langle \sigma'', \nu', \vec{\varepsilon} \to \nu'(\vec{\kappa})\rangle \models [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]\Gamma(e)$
(where $\vec{\kappa}$ are the free variables of $\Gamma(e)$ and $\vec{\varepsilon}$ are fresh)

$\langle \sigma'', \nu', \vec{\varepsilon} \to \nu'(\vec{\kappa})\rangle \models \alpha.c^{i+1}(\delta)$ — By def. of $\models$
(where i is the maximum call height in the above)

$\langle \sigma'', \nu', \vec{\varepsilon} \to \nu'(\vec{\kappa})\rangle \models \alpha.c(\delta)$ — By def. of $\models$

In the above, $\nu', \vec{\varepsilon} \to \nu'(\vec{\kappa})$ is an extension of valuation ν' with mappings from ε_i to $\nu'(\kappa_i)$ for each i. This extension step assumes basic substitution and weakening properties.

The remaining cases are similar, though require applying weakening of ν in evaluation (Lemma A.2). □

The following states a weakening property on valuations (adding of unused mappings) in evaluation, which is needed in the proof above.

Lemma A.2 (Weakening valuations in evaluation)**.**

1. *If* $\nu \vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', u\rangle$ *and* $\nu' \supseteq \nu$*, then* $\nu' \vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', u\rangle$*.*
2. *If* $\nu \vdash t \Downarrow u$ *and* $\nu' \supseteq \nu$*, then* $\nu' \vdash t \Downarrow u$*.*

Proof. By induction on the given derivations. □

A.2 Typing: An Approximation of Checker Evaluation

Theorem A.3 (Typing computes an approximation of time-stamped stores)**.**

Restatement of Theorem 3.9. Assume that each checker definition has been type-checked.

If $\nu \vdash \langle \bar{\sigma}, e\rangle_k \Downarrow \langle \bar{\sigma}', u\rangle$, $\Gamma \vdash e\ \mathsf{ok}$, *and* $\vdash \langle \bar{\sigma}, \nu\rangle : \Gamma + k$,
then $\vdash \langle \bar{\sigma}', \nu'\rangle : \Gamma' + k$ *(for some* $\nu' \supseteq \nu$ *and* $\Gamma' \supseteq \Gamma$*).*

In the above, $\Gamma + k$ *is the function that increments all the levels in* Γ *by* k *(i.e., lifting the function on types).*

Proof. By induction on the first two derivations and case analysis on last one.

Case:
$$\frac{\nu, \beta \to u \vdash \langle [\nu(\alpha) + f \mapsto u]_{\alpha\cdot k} * \bar{\sigma}, e\rangle_k \Downarrow \langle \bar{\sigma}', u\rangle}{\nu \vdash \langle [\nu(\alpha) + f \mapsto u]_{\mathsf{any}} * \bar{\sigma}, \mathrm{let}\ \beta = \alpha\,.\,f\ \mathrm{in}\ e\rangle_k \Downarrow \langle \bar{\sigma}', u\rangle}\ \textsf{ie-read}$$

$$\frac{\{f\langle 0\rangle\} <: \Gamma(\alpha) \qquad \Gamma, \beta : \tau \vdash e\ \mathsf{ok}}{\Gamma \vdash \mathrm{let}\ \beta : \tau = \alpha\,.\,f\ \mathrm{in}\ e\ \mathsf{ok}}\ \textsf{t-read}$$

$$\frac{\vdash \langle \bar{\sigma}, \nu\rangle : \Gamma + k}{\vdash \langle [\nu(\alpha) + f \mapsto u]_{\mathsf{any}} * \bar{\sigma}, \nu\rangle : \Gamma + k}\ \textsf{st-any}$$

$\{f\langle k\rangle\} <: ((\Gamma, \beta : \tau) + k)(\alpha)$	By level increment
$\vdash \langle [\nu(\alpha) + f \mapsto u]_{\alpha \cdot k} * \bar{\sigma}, \nu\rangle : (\Gamma, \beta : \tau) + k$	By $\mathsf{st\text{-}stamped}$ and weakening
$\vdash \langle \bar{\sigma}', \nu'\rangle : \Gamma' + k$ (for some $\nu' \supseteq \nu, \beta \rightarrow u$ and $\Gamma' \supseteq \Gamma, \beta : \tau$)	By i.h.

Case:

$$\frac{\nu \vdash \langle \bar{\sigma}, [\alpha, \delta/\pi, \rho]e\rangle_{k+1} \Downarrow \langle \bar{\sigma}', u\rangle \quad (\pi \,.\, c(\rho) \texttt{ := } e)}{\nu \vdash \langle \bar{\sigma}, \alpha \,.\, c(\delta)\rangle_k \Downarrow \langle \bar{\sigma}', u\rangle}\ \textsf{ie-call}$$

$$\frac{\Gamma(\alpha) - 1 = \tau_0 \quad \Gamma(\delta) - 1 = \tau_1 \quad ((\pi : \tau_0) \,.\, c(\rho : \tau_1) \texttt{ := } e)}{\Gamma \vdash \alpha \,.\, c(\delta)\ \textsf{ok}}\ \textsf{t-call}$$

$\pi : \tau_0, \rho : \tau_1 \vdash e\ \textsf{ok}$	Given
$\Gamma - 1 \vdash [\alpha, \delta/\pi, \rho]e\ \textsf{ok}$	By weakening and substitution
$\vdash \langle \bar{\sigma}, \nu\rangle : \Gamma - 1 + k + 1$	Given
$\vdash \langle \bar{\sigma}', \nu'\rangle : \Gamma' + k + 1$ (for some $\nu' \supseteq \nu$ and $\Gamma' \supseteq \Gamma - 1$)	By i.h.
$\vdash \langle \bar{\sigma}', \nu'\rangle : \Gamma'' + k$ and $\Gamma'' \supseteq \Gamma$	Let $\Gamma'' = \Gamma' + 1$

The remaining cases are similar. $\square$

A.3 Soundness and Termination of Folding

Recall that intuitively, $\nu_\ell \circledcirc \Psi$ is the composition of a valuation ν_ℓ (for a state A_ℓ) with a valuation transformer Ψ to give a valuation for ν_r (for a state A_r). In the case that Ψ is simply a mapping from nodes on the right to nodes on the left, then $\circledcirc$ is simply function composition. In particular, we assume the following property of $\circledcirc$.

Property A.4 (Transformation of valuations)**.**

If $(\sigma, \nu) \in \boldsymbol{\gamma}(M)$, $\Psi(\vec{\alpha_r}) = \vec{\alpha_\ell}$, *and* $\vec{\alpha_\ell}$ *are the free variables of* M,
then $(\sigma, \nu \circledcirc \Psi) \in \boldsymbol{\gamma}([\vec{\alpha_r}/\vec{\alpha_\ell}]M)$.

I write $\Psi(\vec{\alpha_r}) = \vec{\alpha_\ell}$ to indicate some set of symbolic values $\vec{\alpha_r}$ that map to the symbolic values $\vec{\alpha_\ell}$ in Ψ, which may not be unique.

Theorem A.5 (Soundness of comparison in the shape domain)**.**

Restatement of Theorem 4.9. The comparison operation on shape graphs is sound.

If $M_\ell \sqsubseteq^F_\Psi M_r$, $(\sigma, \nu) \in \boldsymbol{\gamma}(M_\ell)$, *and* $\nu \circledcirc \Psi$ *satisfies* F, *then* $(\sigma, \nu \circledcirc \Psi) \in \boldsymbol{\gamma}(M_r)$.

Proof. By induction on the derivation of $M_\ell \sqsubseteq^F_\Psi M_r$.

Case:
$$\dfrac{M_\ell \sqsubseteq^F_\Psi M'_r \qquad (M'_r, F') \in \mathbf{u}_{\alpha_r}(M_r * \alpha_r.c(\delta_r))}{M_\ell \sqsubseteq^{F \wedge F'}_\Psi M_r * \alpha_r.c(\delta_r)}\ \textsf{c-uchk}$$

$\nu \circledcirc \Psi$ satisfies $F \wedge F'$	Given
$(\sigma, \nu \circledcirc \Psi) \in \boldsymbol{\gamma}(M_r)$	By i.h.
$(\sigma, \nu \circledcirc \Psi) \in \boldsymbol{\gamma}(M_r * \alpha_r.c(\delta_r))$	By Lemma 4.8

The case for $\textsf{c-useg}$ is analogous.

Case:
$$\dfrac{\begin{array}{ll}\Psi(\alpha_r) = \alpha_\ell & M_\ell \sqsubseteq^F_\Psi M_r * \alpha'_r.c'(\delta'_r) \\ \Psi(\delta_r) = \delta_\ell \qquad \Psi(\alpha'_r) = \alpha'_\ell & \Psi(\delta'_r) = \delta'_\ell \qquad (\alpha'_r, \delta'_r \text{ fresh})\end{array}}{M_\ell * \alpha_\ell.c(\delta_\ell) \mathrel{*\!\!=} \alpha'_\ell.c'(\delta'_\ell) \sqsubseteq^F_\Psi M_r * \alpha_r.c(\delta_r)}\ \textsf{c-segchk}$$

$(\sigma' * \sigma'', \nu) \in \boldsymbol{\gamma}(M_\ell * \alpha_\ell.c(\delta_\ell) \twoheadleftarrow \alpha'_\ell.c'(\delta'_\ell))$	Given
$(\sigma', \nu) \in \boldsymbol{\gamma}(M_\ell)$	By def. of $\boldsymbol{\gamma}$ and $\models$
$(\sigma'', \nu) \in \boldsymbol{\gamma}(\alpha_\ell.c(\delta_\ell) \twoheadleftarrow \alpha'_\ell.c'(\delta'_\ell))$	By def. of $\boldsymbol{\gamma}$ and $\models$
$(\sigma', \nu \otimes \Psi) \in \boldsymbol{\gamma}(M_r * \alpha'_r.c'(\delta'_r))$	By i.h.
$\sigma' = \sigma''' * \sigma''''$ (for some σ''' and σ'''')	By def. of $\boldsymbol{\gamma}$ and $\models$
$(\sigma''', \nu \otimes \Psi) \in \boldsymbol{\gamma}(M_r)$	By def. of $\boldsymbol{\gamma}$ and $\models$
$(\sigma'''', \nu \otimes \Psi) \in \boldsymbol{\gamma}(\alpha'_r.c'(\delta'_r))$	By def. of $\boldsymbol{\gamma}$ and $\models$
$(\sigma'', \nu \otimes \Psi) \in \boldsymbol{\gamma}(\alpha_r.c(\delta_r) \twoheadleftarrow \alpha'_r.c'(\delta'_r))$	By Property A.4
$(\sigma'' * \sigma'''', \nu \otimes \Psi) \in \boldsymbol{\gamma}(\alpha_r.c(\delta_r))$	By Corollary 3.5
$(\sigma'' * \sigma''' * \sigma'''', \nu \otimes \Psi) \in \boldsymbol{\gamma}(M_r * \alpha_r.c(\delta_r))$	By def. of $\boldsymbol{\gamma}$ and $\models$

The case for c-segseg is analogous. Finally, the cases for c-emp, c-pt, and c-chk are straightforward. ☐

Lemma A.6 (Soundness of fragment rewriting)**.**

Restatement of Lemma 4.12. The fragment rewriting rules preserve inclusion in the concretizations.

If $\langle m_\ell, m_r\rangle \rightsquigarrow_{\Psi}^{F_\ell, F_r} m$, *then*

1. *if* $(\sigma_\ell, \nu_\ell) \in \boldsymbol{\gamma}(m_\ell)$ *and* $\nu_\ell \otimes \Psi_\ell$ *satisfies* F_ℓ, *then* $(\sigma_\ell, \nu_\ell \otimes \Psi_\ell) \in \boldsymbol{\gamma}(m)$.
2. *if* $(\sigma_r, \nu_r) \in \boldsymbol{\gamma}(m_r)$ *and* $\nu_r \otimes \Psi_r$ *satisfies* F_r, *then* $(\sigma_r, \nu_r \otimes \Psi_r) \in \boldsymbol{\gamma}(m)$.

Proof. By case analysis on the derivation of $\langle m_\ell, m_r\rangle \rightsquigarrow_{\Psi}^{F_\ell, F_r} m$.

Case:
$$\frac{\begin{array}{lll}\Psi(\alpha) = (\alpha_\ell, \alpha_r) & \Psi(\alpha') = (\alpha_\ell, \alpha'_r) & m_r \sqsubseteq^F_{\Psi_r} \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c(\delta') \\ \Psi_\ell(\delta) = \delta_\ell & \Psi_\ell(\delta') = \delta_\ell & \end{array}}{\langle \mathsf{emp}, m_r \rangle \rightsquigarrow^{\mathsf{t},F}_{\Psi} \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c(\delta')} \;\textsf{j-waliases}$$

Subcase: left side

$\sigma_\ell = [\cdot]$	By def. of $\boldsymbol{\gamma}$ and $\models$
$(\nu_\ell \otimes \Psi_\ell)(\alpha) = (\nu_\ell \otimes \Psi_\ell)(\alpha')$	As $\Psi_\ell(\alpha) = \alpha_\ell$ and $\Psi_\ell(\alpha') = \alpha_\ell$
$(\nu_\ell \otimes \Psi_\ell)(\delta) = (\nu_\ell \otimes \Psi_\ell)(\delta')$	As $\Psi_\ell(\delta) = \delta_\ell$ and $\Psi_\ell(\delta') = \delta_\ell$
$(\sigma_\ell, \nu_\ell \otimes \Psi_\ell) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c(\delta') \,)$	By def. of $\boldsymbol{\gamma}$ and $\models$

Subcase: right side

$(\sigma_r, \nu_r \otimes \Psi_r) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \,)$	By Theorem 4.9

The remaining cases are similar. For the matching cases j-pt and j-chk, they follow directly from the definition of $\boldsymbol{\gamma}$ and $\models$. The other weakening cases j-wchk and j-wseg require an application of Theorem 4.9 (soundness of comparison in the shape domain). □

Lemma A.7 (Satisfaction of residual constraints)**.** *Satisfaction of residual constraints is closed under backwards rewriting.*

If $(M_\ell \sqcup M_r) \circledast_{F_\ell,F_r} M \rightsquigarrow_\Psi (M'_\ell \sqcup M'_r) \circledast_{F'_\ell,F'_r} M'$, *then*

1. *if* ν_ℓ *satisfies* F'_ℓ, *then* ν_ℓ *satisfies* F_ℓ.
2. *if* ν_r *satisfies* F'_r, ν_r *satisfies* F_r.

As a consequence,

if $(M_\ell \sqcup M_r) \circledast_{F_\ell,F_r} M \rightsquigarrow^*_\Psi (M'_\ell \sqcup M'_r) \circledast_{F'_\ell,F'_r} M'$, *then*

1. *if* ν_ℓ *satisfies* F'_ℓ, *then* ν_ℓ *satisfies* F_ℓ.
2. *if* ν_r *satisfies* F'_r, ν_r *satisfies* F_r.

Proof. Direct by the definition of rewriting ((4.1a) in Definition 4.13). The consequence is proven by induction on the derivation of the multistep rewriting. □

Theorem A.8 (Stabilization of widening in the shape domain)**.**

Restatement of Theorem 4.17. Given any sequence of individual states $\langle E'_n, M'_n, P'_n\rangle_{n\in\mathbb{N}}$, let $\langle E_n, M_n, P_n\rangle_{n\in\mathbb{N}}$ be the sequence defined as follows:

$$\begin{aligned}\langle E_0, M_0, P_0\rangle &\stackrel{\text{def}}{=} \langle E'_0, M'_0, P'_0\rangle \\ \langle E_{n+1}, M_{n+1}, P_{n+1}\rangle &\stackrel{\text{def}}{=} \langle E_n, M_n, P_n\rangle \mathbin{\triangledown} \langle E'_{n+1}, M'_{n+1}, P'_{n+1}\rangle\ .\end{aligned}$$

Then, the sequence $(M_n)_{n\in\mathbb{N}}$ (computed by joins in the shape domain defined in Definition 4.13) is ultimately stationary.

Proof. Let $(M_n)_{n\in\mathbb{N}}$ be defined as above. Next, let a *path* in M_i be defined as a variable x followed by a sequence of field edges where $x \in \mathrm{dom}(E_i)$. Observe that in the fragment rewriting rules (Figure 4.8), a points-to edge can only be present in the upper bound if points-to edges existed in the inputs. Thus, the set of paths in M_{i+1} is contained in the set of paths in M_i, that is,

$$\mathrm{paths}(M_{i+1}) \subseteq \mathrm{paths}(M_i)\ .$$

Thus, the sequence of sets of paths $(\mathrm{paths}(M_n))_{n\in\mathbb{N}}$ converges to some limit. Furthermore, we can define an equivalence relation between paths where two paths are equivalent when they evaluate to the same node (i.e., traversals from their respective variables along their respective field edges end up at the same node). The sequence of equivalence classes of paths defined by this relation must also converge to some limit.

The j-waliases is the only rule that can create a new node in M_{n+1} not present in M_n. However, when it is used, an equivalence class is split, so j-waliases can be applied only finitely many times. After j-waliases is applied for the last time, the set of nodes decreases and converges to some limit. Then, the set of edges must converge. Thus, the sequence $(M_n)_{n\in\mathbb{N}}$ is ultimately stationary. $\square$