Tools and Techniques for the Sound Verification of Low-Level Code

by

Christopher L. Conway

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
January 2011

Clark Barrett—Advisor

Dedication

For Hilleary. And why not?

Acknowledgments

I'd like to express my appreciation to all those without whose mentorship and support I would not have been able to complete this work: Stephen Edwards, Dennis Dams, Kedar Namjoshi, Sriram Rajamani, and, of course, my advisor Clark Barrett. I'm grateful also for the advice and friendship of Al Aho and Amir Pnueli, who helped me through difficult times with wisdom and grace.

I am in debt to all of the members of the Analysis of Computer Systems research group at NYU, who contributed enormously with their feedback, questions, and criticism through every stage of my research. I owe particular thanks to Ittai Balaban, who was—one day, long ago, in a foreign land—the first to suggest I become involved with the ACSys group, and to my office mates Dejan Jovanović and Mina Jeong, who put up with a lot and got very little in return. (Dejan also kindly provided Figure 4.2 free of charge.)

I would like to thank the members of my committee for their time and their valuable feedback: Clark Barrett, Patrick Cousot, Morgan Deters, Ben Goldberg, and Kedar Namjoshi.

Most of all, I owe thanks to my family—especially to my wife, Hilleary—who made all of this possible.

Portions of this thesis are based on work supported by the National Science Foundation under Grant Numbers 0341685 and 0644299, in collaboration with Dennis Dams, Kedar Namjoshi, and Clark Barrett.

Abstract

Software plays an increasingly crucial role in nearly every facet of modern life, from communications infrastructure to control systems in automobiles, airplanes, and power plants. To achieve the highest degree of reliability for the most critical pieces of software, it is necessary to move beyond *ad hoc* testing and review processes towards verification—to prove using *formal* methods that a program exhibits exactly those behaviors allowed by its specification and no others.

A significant portion of the existing software infrastructure is written in low-level languages like C and C++. Features of these languages present significant verification challenges. For example, unrestricted pointer manipulation means that we cannot prove even the simplest properties of programs without first collecting precise information about potential aliasing relationships between variables.

In this thesis, I present several contributions to the field of program verification. The first is a general framework for combining program analyses that are only conditionally sound. Using this framework, I show it is possible to design a sound verification tool that relies on a separate, previously-computed pointer analysis.

The second contribution of this thesis is Cascade, a multi-platform, multi-paradigm framework for verification. Cascade includes support for precise analysis of low-level C code, as well as for higher-level languages such as SPL.

Finally, I describe a novel technique for the verification of datatype invariants in low-level systems code. The programmer provides a high-level specification for a low-level implementation in the form of inductive datatype declarations and code assertions. The connection between the high-level semantics and the implementation code is then checked using bit-precise reasoning. An implementation of this datatype verification technique is available as a Cascade module.

Contents

List of Figures

List of Tables

Introduction

Software plays an increasingly crucial role in nearly every facet of modern life. Estimates of the total cost imposed by software failures range from nearly USD 60 billion annually in the United States [64] to more than USD 1 trillion globally [61]. Software bugs have been identified as the cause of catastrophic failures, include: the loss of an Ariane 5 rocket, which cost more than USD 370 million [23]; the 2003 blackout in the United States and Canada, which left more than 50 million people without power [26]; and the Therac-25 radiation therapy machine malfunction, which killed at least three patients [5].

For certain pieces of critical software, the *ad hoc* methods of software engineering—documentation, code review, testing—are not sufficient. It is necessary to apply *formal* methods—tools and techniques that allow us to *prove* that the code is correct. In particular we are interested in *sound verification*—methods that are guaranteed to find a bug, if one exists. The use of sound analysis means we can prove the absence of errors in a program. On the other hand, a sound analysis may not be *complete*, in the sense that it may produce *false alarms* (i.e., spurious errors) on correct programs.

In recent years, great advances have been made in verification technology. Tools such as SLAM/SDV [6] have brought software model checking to the mainstream.

The ASTRÉE static analyzer [16] has been used to verify control systems in Airbus aircraft [19]. However, there are still significant open challenges in verification, particularly as applied to low-level systems software written in languages like C [4] and C++ [35]. These languages have features like unrestricted pointer manipulation that render even the simplest analysis problems undecidable [39].

Due to the central role that pointers play in low-level code, it is impossible to do any precise analysis of such programs without first obtaining information about pointer relationships, e.g., from an *aliasing* or *points-to* analysis [24, 66, 3, 63, 18]. This means that any sound verification tool must rely on the results of a pointer analysis, which may not itself be sound. This seems to present a circular dilemma: how can a sound analysis rely on a potentially unsound input?

In the first two chapters of this thesis, I show how this dilemma can be resolved. I present a framework for describing *conditionally sound* analyses and show that a sound analysis can be built that relies on the results of a conditionally sound prior analysis. The idea of a conditionally sound analysis is not novel—it is present in the Cousots' work on abstract interpretation dating to the 1970s [15, 16, 17]. However, the framework presented here is a convenient way to capture the behavior of several interesting analyses directly and the combination theorem I present lays out clearly the relationship between conditionally sound analyses and subsequent analyses that rely on their results.

In the second chapter of this thesis, I apply the conditional soundness framework to pointer analysis, showing that a set of points-to analyses similar to and sharing the soundness properties of commonly-used flow-sensitive and insensitive analyses—such as those of Emami, Ghiya, and Hendren [24]; Wilson and Lam [66]; Andersen [3]; Steensgaard [63]; and Das [18]—provide results that are sound for any *memory-safe execution* of a program. This statement is both stronger and more precise than the traditional statement that such analyses are sound for "well-behaved" programs. I also show that this condition on the soundness of the analysis is tight: given certain reasonable assumptions, no pointer analysis can be sound under any weaker condition. This more precise characterization of a points-to analysis, along with the combination theorem for conditional analyses, shows that the combination of an independent points-to analysis with a memory safety analysis is conditionally sound.

In the third chapter, I describe CASCADE, a multi-platform, multi-paradigm framework for verification developed at NYU. CASCADE is suitable for a broad class of languages, ranging from low-level implementation languages like C to high-level modeling languages like SPL [47, 48]. CASCADE is open source, extensible, and available for free download from the NYU Analysis of Computer Systems group website.

In the final chapter, I describe an application of CASCADE to a real, low-level verification challenge. I propose a novel technique for the verification of datatype invariants in low-level systems code, one which fuses the power of higher-level datatypes with the convenience and efficiency of legacy code. The technique makes use of the theories of inductive datatypes, bit vectors, arrays, and uninterpreted functions in the CVC3 SMT solver[7] to encode the relationship between the high-level and low-level semantics. High-level datatype assertions are then checked using bit-precise reasoning.

Taken together, the results described in this thesis represent a modest, but non-trivial improvement on the state of the art in software verification.

Chapter 1

A Program Analysis Framework

To present program analysis in a formal setting, we use the framework of abstract interpretation [13]. A full syntax of program statements is given in Section 2.1. For now, we are concerned only with the relationship between concrete and abstract interpretations. We omit any discussion of techniques (such as widening and extrapolation) which serve to make program analyses finite and computable—we are concerned solely with issues of soundness.

1.1 Program Semantics and Analyses

A *partial order* $(S, \sqsubseteq)$ is a pair, where S is a set and $\sqsubseteq$ is a reflexive, transitive, and antisymmetric binary relation on S. When the meaning is clear, we overload S to refer to both a partial order and its underlying set of states and we use $\sqsubseteq$ or $\sqsubseteq_S$ to refer to its ordering relation.

Let S be a partial order and let S' be a subset of S. S' is *downward closed* if a is in S' whenever b is in S' and $a \sqsubseteq b$. An element $a \in S$ is an *upper bound of* S' if $b \sqsubseteq a$ for every element $b \in S'$; a is the *least* upper bound of S', denoted

$\sqcup S'$, if a is an upper bound of S' and $a \sqsubseteq b$ for every upper bound b of S'. S' is a *directed subset* if every pair of elements in S' has an upper bound in S'. S is a *complete partial order* if each of its directed subsets has a least upper bound and there exists a least element $\perp \in S$.

Let $F : X \to Y$ be a function from the partial order X to the partial order Y. F is *monotone* if $F(a) \sqsubseteq_Y F(b)$ whenever $a \sqsubseteq_X b$.

Let $F : S \to S$ be a function on the partial order S. F is *decreasing* if $F(a) \sqsubseteq a$, for all $a \in S$. An element $a \in S$ is a *fixed point of* F if $F(a) = a$; a is the *least* fixed point of F, denoted $\mathrm{lfp}(F)$, if a is a fixed point a of F and $a \sqsubseteq b$ for every fixed point b of F. F is a *downward closure for* $S' \subseteq S$ if: F is decreasing and monotone; $F(a)$ is in S', for all $a \in S$; and every element of S' is a fixed point of F.

Let S be a complete partial order with bottom element $\perp$ and $F : S \to S$ a monotone function. The *iterates of* F are the sequence $X_0, X_1, \ldots, X_\omega, X_{\omega+1}, \ldots$ defined as follows:

$$\begin{aligned} X_0 &= \perp \\ X_{\alpha+1} &= F(X_\alpha), && \text{for successor ordinal } \alpha + 1 \\ X_\lambda &= \sqcup\{X_\alpha \mid \alpha < \lambda\}, && \text{for limit ordinal } \lambda \end{aligned}$$

Not every function has fixed points; nor does every function with fixed points necessarily have a least fixed point. However, monotone functions over complete partial orders do have least fixed points and, furthermore, the least fixed point can be reached by iteration.

Bourbaki's[1] Fixed Point Theorem [9]. *Let* $(X, \sqsubseteq)$ *be a complete partial order and* $F : X \to X$ *a monotone function.* F *has a least fixed point and it is equal to the least upper bound of the iterates of* F*, i.e.,* $\mathrm{lfp}(F) = \bigsqcup_{\alpha \geq 0} X_\alpha$

Complete partial orders give us a very general setting for reasoning about the correctness of analyses. We will define the semantics of a program as an element of a complete partial order—the least fixed point of a monotone function F computing the reachable states of the program. Likewise, the result of an analysis will be an element of a complete partial order—the least fixed point of a monotone function G over-approximating the reachable states of the program. The standard Abstraction Theorem tell us that, if F "abstracts" G, then $\mathrm{lfp}(F)$ "abstracts" $\mathrm{lfp}(G)$.

Abstraction Theorem [13]. *Let* X *and* Y *be complete partial orders and* $F : X \to X$*,* $G : Y \to Y$*,* $\gamma : Y \to X$ *monotone functions. If* $F(x) \sqsubseteq_X \gamma(G(y))$ *whenever* $x \sqsubseteq_X \gamma(y)$*, then* $\mathrm{lfp}(F) \sqsubseteq_X \gamma(\mathrm{lfp}(G))$.

Now we will define our analysis framework. Let C be a distinguished set of *concrete states*. A *domain* (D, γ) is a pair, where D is a set of *abstract states* and $\gamma : D \to 2^C$ is a *concretization function*. When the meaning is clear, we overload D to refer both to a domain and to its underlying set of states and use γ_D to refer to the concretization function. We lift γ_D to sets of states: $\gamma_D(D') = \bigcup_{d \in D'} \gamma_D(d)$, where $D' \subseteq D$.

The *concrete domain* D_C is given by the pair (C, γ_C), where γ_C is the trivial

---

[1] The history of fixed point theorems is long and tangled; variants of the theorem given here have been variously attributed to Knaster, Tarski, Kleene, and Scott [42, 37]. Bourbaki's theorem [9] appears to be the first that is substantially similar to the theorem stated here—it requires F to be "expansive" (i.e., $x \sqsubseteq F(x)$, for all x), but the proof is easily modified for the case where F is monotone. Cousot and Cousot prove a theorem [14, Corollary 3.3] which is essentially identical to the version we use—it assumes the domain is a complete lattice, but the proof does not make use of the top element or greatest lower bounds; hence it applies equally to complete partial orders.

concretization function: $\gamma_C(c) = \{c\}$. We say a set $D' \subseteq D$ *over-approximates* $C' \subseteq C$ iff $\gamma_D(D') \supseteq C'$. Similarly, a function $F_D : D \to 2^D$ *over-approximates* $F_C : C \to 2^C$ iff $F_D(d)$ over-approximates $F_C(c)$ whenever c is in $\gamma_D(d)$.

Every domain D is associated with the complete partial order 2^D, ordered by inclusion. The function γ_D, lifted to 2^D, is trivially monotone.

We define the soundness of a program interpretation in terms of a *collecting semantics*. Given a (concrete or abstract) domain D, we will define a *semantic operator* $[\![\cdot]\!]$ which maps a program $\mathcal{P}$ to to a set $[\![\mathcal{P}]\!] \subseteq D$ of *reachable states*. The semantics $[\![\mathcal{P}]\!]$ is defined in terms of *semantic interpretations over* D: a set $\mathcal{I}[\mathcal{P}] \subseteq D$ of *initial states* and a *transfer function* $\mathcal{F}[\mathcal{P}] : D \to 2^D$. We lift $\mathcal{F}[\mathcal{P}]$ to sets of states: $\mathcal{F}[\mathcal{P}](D') = \bigcup_{d \in D'} \mathcal{F}[\mathcal{P}](d)$, where $D' \subseteq D$—so lifted, $\mathcal{F}[\mathcal{P}]$ is trivially continuous.

An *analysis* $\mathcal{A}$ is represented as a tuple $(D, \mathcal{I}, \mathcal{F})$, where D is a domain and $\mathcal{I}$ and $\mathcal{F}$ are semantic interpretations over D. We use $D_{\mathcal{A}}$, $\mathcal{I}_{\mathcal{A}}$, and $\mathcal{F}_{\mathcal{A}}$ to denote the constituents of an analysis $\mathcal{A}$ and $\gamma_{\mathcal{A}}$ to denote the concretization function of the domain $D_{\mathcal{A}}$.

Definition 1. Let $\mathcal{A} = (D, \mathcal{I}, \mathcal{F})$ be an analysis. The *semantics* $[\![\cdot]\!]_{\mathcal{A}}$ *w.r.t.* $\mathcal{A}$ maps a program $\mathcal{P}$ to a subset of D, the *reachable states in* $\mathcal{P}$ *w.r.t.* $\mathcal{A}$:

$$[\![\mathcal{P}]\!]_{\mathcal{A}} = \mathrm{lfp}(\mathbf{F}_{\mathcal{A}}[\mathcal{P}]), \quad \text{where } \mathbf{F}_{\mathcal{A}}[\mathcal{P}] = \lambda S.\ \mathcal{I}[\mathcal{P}] \cup \mathcal{F}[\mathcal{P}](S)$$

Note that $\mathbf{F}_{\mathcal{A}}[\mathcal{P}]$ is monotone and thus, by Bourbaki's Theorem, the least fixed point exists and the semantics is well-defined.

To judge the soundness of an analysis, we need a concrete semantics against which it can be compared. We assume that a *concrete analysis* $\mathcal{C} = (D_{\mathcal{C}}, \mathcal{I}_{\mathcal{C}}, \mathcal{F}_{\mathcal{C}})$

is given. The concrete analysis uniquely defines a *concrete semantics* $[\![\cdot]\!]_{\mathcal{C}}$.

Definition 2. An analysis $\mathcal{A}$ is *sound* iff for every program $\mathcal{P}$, $[\![\mathcal{P}]\!]_{\mathcal{A}}$ over-approximates $[\![\mathcal{P}]\!]_{\mathcal{C}}$ (i.e., $\gamma_{\mathcal{A}}([\![\mathcal{P}]\!]_{\mathcal{A}}) \supseteq [\![\mathcal{P}]\!]_{\mathcal{C}}$).

By the Abstraction Theorem, it is sufficient for $\mathcal{A}$ to have sound semantic interpretations.

Definition 3. Let $\mathcal{I}_D$ be a semantic interpretation over a domain D. $\mathcal{I}_D$ is *sound* iff for every program $\mathcal{P}$, $\mathcal{I}_D[\mathcal{P}]$ over-approximates $\mathcal{I}_{\mathcal{C}}[\mathcal{P}]$. $\mathcal{F}_D$ is *sound* iff for every program $\mathcal{P}$, $\mathcal{F}_D[\mathcal{P}]$ over-approximates $\mathcal{F}_{\mathcal{C}}[\mathcal{P}]$.

Theorem 1.1. *Let $\mathcal{A}$ be an analysis. If $\mathcal{I}_{\mathcal{A}}$ and $\mathcal{F}_{\mathcal{A}}$ are sound, then $\mathcal{A}$ is sound.*

Proof. Theorem 1.4, below, generalizes this Theorem. □

1.2 Conditional Soundness

So far, we have defined a style of analysis which is *unconditionally sound*, mirroring the traditional approach to abstract interpretation. We will primarily be interested in analyses that are sound only under certain assumptions about the behavior of the program analyzed. To address this, we introduce the notion of *conditional soundness with respect to a predicate θ*. An analysis will be *θ-sound* if it over-approximates the concrete states of a program that are reachable via *only θ-states.* We first define a semantics restricted to θ.

Definition 4. Let $\mathcal{A} = (D, \mathcal{I}, \mathcal{F})$ be an analysis and θ a predicate on D (we view the predicate θ, equivalently, as a subset of D). The *θ-restricted semantics* $[\![\cdot]\!]_{\mathcal{A}}\!\downarrow_{\theta}$ *w.r.t.* $\mathcal{A}$ maps a program $\mathcal{P}$ to a subset of D, the *θ-reachable states in $\mathcal{P}$ w.r.t. $\mathcal{A}$*:

$$[\![\mathcal{P}]\!]_{\mathcal{A}}\!\downarrow_{\theta} = \mathrm{lfp}(\mathbf{F}_{\mathcal{A}}[\mathcal{P}] \circ \mathbf{G}_{\theta}), \quad \text{where } \mathbf{G}_{\theta} = \lambda S.\ \theta \cap S$$

$\mathbf{F}_{\mathcal{A}}[\mathcal{P}]$ is the same as in Definition 1.

Note that $[\![\mathcal{P}]\!]_{\mathcal{A}}{\downarrow_\theta}$ may include non-θ states—the range of $\mathbf{F}_{\mathcal{A}}[\mathcal{P}] \circ \mathbf{G}_\theta$ is not restricted to θ—but those states will not have an "successors" in fixed point computation. The θ-restricted semantics gives us a lower bound for the approximation of a θ-sound analysis.

Definition 5. Let $\mathcal{A}$ be an analysis and θ a predicate on C. $\mathcal{A}$ is *θ-sound* iff for every program $\mathcal{P}$, $[\![\mathcal{P}]\!]_{\mathcal{A}}$ over-approximates $[\![\mathcal{P}]\!]_{\mathcal{C}}{\downarrow_\theta}$.

Note that an unconditionally sound analysis is also θ-sound for *any* θ. More generally, any θ-sound analysis is also φ-sound, for any φ stronger than θ.

This notion of conditional soundness does not just give us a more precise statement of the behavior of certain analyses—it provides us with a sufficient condition to show an analysis *proves the absence of error states*. This is a consequence of the following general property of fixed points.

Theorem 1.2. *Let X be a complete partial order; $F : X \to X$ a monotone function; S a downward closed subset of X; and $G : X \to S$ a downward closure for S. An element x of S is the least fixed point of $F \circ G$ iff x is the least fixed point of F.*

Proof. Since F and G are both monotone, $F \circ G$ is monotone. Hence, both F and $F \circ G$ have least fixed points, by Bourbaki's Theorem.

Let x be an element of S. Since G is a downward closure for S, x is a fixed point of G. Hence, x is a fixed point of $F \circ G$ iff x is a fixed point of F.

Assume that $\mathrm{lfp}(F)$ is in S. Then $\mathrm{lfp}(F)$ is a fixed point for $F \circ G$. Hence, $\mathrm{lfp}(F \circ G) \sqsubseteq \mathrm{lfp}(F)$. Since S is downward closed, $\mathrm{lfp}(F \circ G)$ is also in S. Thus,

$\mathrm{lfp}(F \circ G)$ is fixed point of F and so $\mathrm{lfp}(F) \sqsubseteq \mathrm{lfp}(F \circ G)$. Therefore $\mathrm{lfp}(F) = \mathrm{lfp}(F \circ G)$

A symmetric argument applies if we assume $\mathrm{lfp}(F \circ G)$ is in S. □

The general theorem applies in our setting, where the downward closed subset S is defined by the predicate θ.

Theorem 1.3. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ a θ-sound analysis. If there are no reachable non-θ states in $\mathcal{P}$ w.r.t. $\mathcal{A}$, then there are no reachable concrete non-θ states in $\mathcal{P}$ (i.e., if $\gamma_A([\![\mathcal{P}]\!]_{\mathcal{A}}) \subseteq \theta$, then $[\![\mathcal{P}]\!]_{\mathcal{C}} \subseteq \theta$).*

Proof. Since $\mathcal{A}$ is θ-sound, and thus $[\![\mathcal{P}]\!]_{\mathcal{C}}\!\downarrow_\theta \subseteq \gamma_A([\![\mathcal{P}]\!]_{\mathcal{A}})$, we have $[\![\mathcal{P}]\!]_{\mathcal{C}}\!\downarrow_\theta \subseteq \theta$. Hence, it is sufficient to show $[\![\mathcal{P}]\!]_{\mathcal{C}} = [\![\mathcal{P}]\!]_{\mathcal{C}}\!\downarrow_\theta$. This follows from Theorem 1.2, taking $F = \mathbf{F}_{\mathcal{C}}[\mathcal{P}]$, $S = 2^\theta$ and $G = \mathbf{G}_\theta$. □

By the Abstraction Theorem, it sufficient for $\mathcal{A}$ to have a θ-sound transfer function.

Definition 6. Let $\mathcal{F}_D$ be a transfer function over domain D. $\mathcal{F}_D$ is *θ-sound* iff for any program $\mathcal{P}$, $\mathcal{F}_D[\mathcal{P}](D')$ over-approximates $\mathcal{F}_{\mathcal{C}}[\mathcal{P}](C')$ whenever D' over-approximates C' *and* $C' \subseteq \theta$.

Theorem 1.4. *Let $\mathcal{A}$ be an analysis. If $\mathcal{I}_{\mathcal{A}}$ is sound and $\mathcal{F}_{\mathcal{A}}$ is θ-sound, then $\mathcal{A}$ is θ-sound.*

Proof. Let $\mathcal{P}$ be a program. By the Abstraction Theorem, taking $F = \mathbf{F}_{\mathcal{C}}[\mathcal{P}] \circ \mathbf{G}_\theta$, $G = \mathbf{F}_{\mathcal{A}}$, and $\gamma = \gamma_{\mathcal{A}}$, it is sufficient to show $\mathbf{F}_{\mathcal{A}}[\mathcal{P}]$ over-approximates $\mathbf{F}_{\mathcal{C}}[\mathcal{P}] \circ \mathbf{G}_\theta$.

Let $C' \subseteq C$ and $D' \subseteq D_{\mathcal{A}}$ such that $C' \subseteq \gamma_{\mathcal{A}}(D')$.

Then,

$$\begin{aligned}
(\mathbf{F}_{\mathcal{C}}[\mathcal{P}] \circ \mathbf{G}_{\theta})(C') &= \mathcal{I}_{\mathcal{C}}[\mathcal{P}] \cup \mathcal{F}_{\mathcal{C}}[\mathcal{P}](\theta \cap C') \\
&\subseteq \gamma_{\mathcal{A}}(\mathcal{I}_{\mathcal{A}}[\mathcal{P}]) \cup \gamma_{\mathcal{A}}(\mathcal{F}_{\mathcal{A}}[\mathcal{P}](D')) && (\mathcal{I}_{\mathcal{A}} \text{ sound, } \mathcal{F}_{\mathcal{A}}\ \theta\text{-sound}) \\
&\subseteq \gamma_{\mathcal{A}}(\mathcal{I}_{\mathcal{A}}[\mathcal{P}] \cup \mathcal{F}_{\mathcal{A}}[\mathcal{P}](D')) && (\gamma_{\mathcal{A}} \text{ monotone}) \\
&= \gamma_{\mathcal{A}}(\mathbf{F}_{\mathcal{A}}[\mathcal{P}](D'))
\end{aligned}$$

Thus, $\mathbf{F}_{\mathcal{A}}[\mathcal{P}]$ over-approximates $\mathbf{F}_{\mathcal{C}}[\mathcal{P}] \circ \mathbf{G}_{\theta}$. □

Note 1.1. Theorem 1.4 generalizes Theorem 1.1: If $\mathcal{F}_{\mathcal{A}}$ is unconditionally sound, then it is θ-sound for any θ. Hence, if $\mathcal{I}_{\mathcal{A}}$ and $\mathcal{F}_{\mathcal{A}}$ are unconditionally sound, $\mathcal{A}$ is unconditionally sound. □

1.3 Parameterized Analysis

Having defined a precise notion of conditional soundness, we now consider how the results of a θ-sound analysis can be used to refine a second analysis. Suppose that $\mathcal{A}$ is an analysis and we have already computed the set of reachable states $[\![\mathcal{P}]\!]_{\mathcal{A}}$. We may wish to use the information present in $[\![\mathcal{P}]\!]_{\mathcal{A}}$ to refine a second analysis over a different domain B. For example, we could use the reduced product construction [15] to form a new domain over a subset of $D_{\mathcal{A}} \times B$ including only those states (a, b) where a is in $[\![\mathcal{P}]\!]_{\mathcal{A}}$ and the states a and b "agree" (e.g., $\gamma_{\mathcal{A}}(a) \cap \gamma_B(d) \neq \emptyset$).

Traditional methods for combining analyses take a "white box" approach—e.g., Cousot and Cousot [15] assume that the analyses will be run in unison, allowing a precise combined analysis to be derived from the two component analyses; Lerner et al. [45] assume that analyses can be run in parallel, one step at a time. In contrast, we will assume that any prior analysis is a black box: we have access to its result (in the form of a set of reachable abstract states), its domain (which allows us to interpret the result), and some (possibly conditional) soundness guarantee. This naturally models the use of off-the-shelf program analyses to provide refinement advice.

We will define such a refinement in terms of a parameterized analysis which produces a new, refined analysis from the results of a prior analysis. An *analysis generator* $\widetilde{\mathcal{G}}$ is a tuple $(D, E, \widetilde{\mathcal{I}}, \widetilde{\mathcal{F}})$ where: D and E are domains (the *input* and *output* domains, respectively) and $\widetilde{\mathcal{I}}$ and $\widetilde{\mathcal{F}}$ are *parameterized interpretations* mapping a set of states $D' \subseteq D$ to semantic interpretations $\widetilde{\mathcal{I}}\langle D' \rangle$ and $\widetilde{\mathcal{F}}\langle D' \rangle$ over E. We denote by $\widetilde{\mathcal{G}}\langle D' \rangle$ the analysis over E defined by the parameterized interpretations on input D': $\widetilde{\mathcal{G}}\langle D' \rangle = (E, \widetilde{\mathcal{I}}\langle D' \rangle, \widetilde{\mathcal{F}}\langle D' \rangle)$. As one might expect, the soundness of $\widetilde{\mathcal{G}}\langle D' \rangle$ depends on the input D'.

Definition 7. An analysis generator $\widetilde{\mathcal{G}}$ with input domain D is *sound* iff for every set of states $D' \subseteq D$, $\widetilde{\mathcal{G}}\langle D' \rangle$ is θ-sound with $\theta = \gamma_D(D')$.

Given an analysis generator, it is natural to consider the analysis formed by composing the generator with an analysis over its input domain. If $\mathcal{A}$ is an analysis and $\widetilde{\mathcal{G}}$ is an analysis generator with input domain $D_{\mathcal{A}}$ (i.e., the input domain of $\widetilde{\mathcal{G}}$ is the underlying domain of $\mathcal{A}$), the *composed analysis* $\widetilde{\mathcal{G}} \circ \mathcal{A}$ is defined by providing the result of $\mathcal{A}$ as a parameter to $\widetilde{\mathcal{G}}$ (i.e., $\widetilde{\mathcal{G}} \circ \mathcal{A} = \widetilde{\mathcal{G}}\langle [\![\mathcal{P}]\!]_{\mathcal{A}} \rangle$). An important property of the composed analysis is preservation of soundness. This arises naturally from a transitivity property of the least fixed points of composed functions.

Theorem 1.5. *Let X be a complete partial order; $F : X \to X$ a monotone function; S a subset of X; T and U downward closed subsets of X; and $G : X \to S$*

and $H : X \to T$ *downward closures for, respectively,* S *and* T*. If* $\mathrm{lfp}(F \circ G)$ *is in* T *and* $\mathrm{lfp}(F \circ H)$ *is in* U*, then* $\mathrm{lfp}(F \circ G)$ *is in* U*.*

Proof. Since U is downward closed, it suffices to show $\mathrm{lfp}(F \circ G) \sqsubseteq \mathrm{lfp}(F \circ H)$. Let $X_0, X_1, \ldots$ be the iterates of $\mathrm{lfp}(F \circ G)$ and $Y_0, Y_1, \ldots$ the iterates of $\mathrm{lfp}(F \circ H)$. By Bourbaki's Theorem, it suffices to show that $X_\alpha \sqsubseteq Y_\alpha$ for all ordinals α. We proceed by induction.

Trivially, $X_0 = Y_0$.

Assume $X_\alpha \sqsubseteq Y_\alpha$ for some ordinal α. Note that X_α is in T, since $X_\alpha \sqsubseteq \mathrm{lfp}(F \circ G)$, $\mathrm{lfp}(F \circ G)$ is in T, and T is downward closed; and $F \circ H$ is monotone, since both F and H are monotone. Thus,

$$\begin{aligned} X_{\alpha+1} &= (F \circ G)(X_\alpha) \\ &\sqsubseteq F(X_\alpha) && (G \text{ decreasing, } F \text{ monotone}) \\ &= (F \circ H)(X_\alpha) && (H \text{ a downward closure for } T) \\ &\sqsubseteq (F \circ H)(Y_\alpha) && (F \circ H \text{ monotone}) \\ &= Y_{\alpha+1} \end{aligned}$$

Now, let λ be a limit ordinal and assume $X_\alpha \sqsubseteq Y_\alpha$ for all ordinals $\alpha < \lambda$. Then,

$$X_\lambda = \bigsqcup_{\alpha<\lambda} X_\alpha \sqsubseteq \bigsqcup_{\alpha<\lambda} Y_\alpha = Y_\lambda$$

This shows $\mathrm{lfp}(F \circ G) \sqsubseteq \mathrm{lfp}(F \circ H)$ □

The general theorem applies in our setting, where the set S is defined by the predicate θ and T is determined by the result of the input analysis.

Theorem 1.6. *If $\widetilde{\mathcal{G}}$ is sound and $\mathcal{A}$ is θ-sound, then the composed analysis $\widetilde{\mathcal{G}} \circ \mathcal{A}$ is θ-sound.*

Proof. Let B be the output domain of $\widetilde{\mathcal{G}}$ and $\mathcal{P}$ a program. Let $\psi = \gamma_{\mathcal{A}}(\llbracket \mathcal{P} \rrbracket_{\mathcal{A}})$ and $\varphi = \gamma_B(\llbracket \mathcal{P} \rrbracket_{\widetilde{\mathcal{G}} \circ \mathcal{A}})$. Applying Theorem 1.5 with $F = \mathbf{F}_{\mathcal{C}}[\mathcal{P}]$, $S = 2^{\theta}$, $T = 2^{\psi}$, $U = 2^{\phi}$, $G = \mathbf{G}_{\theta}$, and $H = \mathbf{G}_{\psi}$, we have $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\theta} = \mathrm{lfp}(\mathbf{F}_{\mathcal{C}}[\mathcal{P}] \circ \mathbf{G}_{\theta}) \subseteq \gamma_B(\llbracket \mathcal{P} \rrbracket_{\widetilde{\mathcal{G}} \circ \mathcal{A}})$. Hence, $\widetilde{\mathcal{G}} \circ \mathcal{A}$ is θ-sound. ☐

It may be helpful to think about Theorem 1.6 informally. The conditional semantics $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\theta}$ is the set of states that are reachable in $\mathcal{P}$ via only θ states. If a state c is in $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\theta}$, then there is some sequence $c_0, c_1, \ldots, c_k$ of concrete states from $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\theta}$ such that:

- c_k is equal to c;
- c_i is in $\mathcal{F}_{\mathcal{C}}[\mathcal{P}](c_{i-1})$, for each $0 < i \leq k$;
- c_0 is in $\mathcal{I}_{\mathcal{C}}[\mathcal{P}]$; and
- each state $c_0, c_1, \ldots, c_{k-1}$ is in θ.

This guarantees that each c_i will appear in the ith iterate of $\mathbf{F}_{\mathcal{C}}[\mathcal{P}]$.

Since $\mathcal{A}$ is θ-sound, we know that $\llbracket \mathcal{P} \rrbracket_{\mathcal{A}}$ over-approximates $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\theta}$. Hence, each of the states $c_0, c_1, \ldots, c_k$ is in $\gamma_{\mathcal{A}}(\llbracket \mathcal{P} \rrbracket_{\mathcal{A}})$ and, thus, each c_i is in $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\gamma_{\mathcal{A}}(\llbracket \mathcal{P} \rrbracket_{\mathcal{A}})}$. In particular, the state c (i.e., c_k) is in $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\gamma_{\mathcal{A}}(\llbracket \mathcal{P} \rrbracket_{\mathcal{A}})}$.

Since $\widetilde{\mathcal{G}}$ is sound, we know that $\llbracket \mathcal{P} \rrbracket_{\widetilde{\mathcal{G}}\langle \llbracket \mathcal{P} \rrbracket_{\mathcal{A}} \rangle}$ over-approximates $\llbracket \mathcal{P} \rrbracket_{\mathcal{C}\downarrow\gamma_{\mathcal{A}}(\llbracket \mathcal{P} \rrbracket_{\mathcal{A}})}$. Hence, c is in $\gamma_B(\llbracket \mathcal{P} \rrbracket_{\widetilde{\mathcal{G}}\langle \llbracket \mathcal{P} \rrbracket_{\mathcal{A}} \rangle})$.

Chapter 2

Pointer Analysis

In this chapter, we show that a set of points-to analyses similar to and sharing the soundness properties of commonly-used flow-sensitive and insensitive analyses—such as those of Emami, Ghiya, and Hendren [24]; Wilson and Lam [66]; Andersen [3]; Steensgaard [63]; and Das [18]—provide results that are sound for any *memory-safe execution* of a program. This statement is both stronger and more precise than the traditional statement that such analyses are sound for "well-behaved" programs.

This more precise characterization of a points-to analysis, along with the combination theorem for conditional analyses, shows that the combination of an independent points-to analysis with a memory safety analysis is conditionally sound. The soundness result guarantees that the *absence* of errors can be proved. Conversely, for a program with memory errors, at least one representative error—but not necessarily all errors—along any unsafe execution will be detected.

$$n \in \mathbb{Z} \qquad \mathtt{x}, \mathtt{y} \in \mathit{Vars}$$

$$\begin{aligned} L \in \mathit{Lvals} &::= \ \mathtt{x} \mid \mathtt{*x} \\ E \in \mathit{Exprs} &::= \ L \mid n \mid \mathtt{x} \oplus \mathtt{y} \mid \mathtt{x} \trianglelefteq \mathtt{y} \mid \mathtt{\&x} \\ S \in \mathit{Stmts} &::= \ L \ \mathtt{:=} \ E \mid \mathtt{[}E\mathtt{]} \end{aligned}$$

Figure 2.1: Grammar for a minimal C-like language.

2.1 Concrete Semantics

To make precise statements about program analyses requires a concrete program semantics. We will define the semantics of the little language presented in Fig. 2.1. The semantics of the language is chosen to model the requirements of ANSI/ISO C [36] without making implementation-specific assumptions. Undefined or implementation-defined behaviors are modeled with explicit nondeterminism. Note that an ANSI/ISO-compliant C compiler is free to implement undefined behaviors in a specific, deterministic manner. By modeling undefined behaviors using nondeterminism, the soundness statements made about each analysis apply to *any* standard-compliant compilation strategy.

The most important features of C that we exclude here are fixed-size integer types, narrowing casts, dynamic memory allocation, and functions.[1] We also ignore the "strict aliasing" rule [36, §6.5]. Each of these can be handled, at the cost of a higher degree of complexity in our definitions.

The syntactic classes of variables, lvalues, expressions, and statements, are defined in Fig. 2.1. We use n to represent an integer constant and `x` and `y` to

---

[1] The omission of dynamic allocation in the discussion of points-to analysis and memory safety may seem an over-simplification. However, it is not essential to our purpose here. Points-to analyses typically handle dynamic allocation by treating each allocation site as if it were the static declaration of a global array of unknown size.

represent arbitrary variables. We use $\oplus$ to represent an arbitrary binary arithmetic operator and $\trianglelefteq$ to represent a relational operator. Pointer operations include arithmetic, indirection (`*`), and address-of (`&`). Statements include assignments and tests (`[`E`]`, where E is an expression).

Variables in our language are viewed as arrays of memory cells. Each cell may hold either an unbounded integer or a pointer value. The only type information present is the allocated size of each variable—the "type system" merely maps variables to their sizes and provides no safety guarantees.

A *program* $\mathcal{P}$ is a tuple $(\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$, where: $\mathcal{V} \subseteq \mathit{Vars}$ is a finite set of *program variables*; $\Gamma : \mathcal{V} \rightarrow \mathbb{Z}^+$ is a *typing environment* mapping a variable to its allocated size (a non-negative integer); $\mathcal{L}$ is a finite set of *program points*; $\mathcal{S} \subseteq \mathit{Stmts}$ is a finite set of *program statements* whose variables are from $\mathcal{V}$; $\tau \subseteq \mathcal{L} \times \mathcal{S} \times \mathcal{L}$ is a *transition relation*; and $en \in \mathcal{L}$ is a distinguished *entry point*. In the following, we assume a fixed program $\mathcal{P} = (\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$.

Example 2.1. Figure 2.2(b) gives a fragment of the program representation for the code in Fig. 2.2(a), corresponding to the function `bad` . We have introduced temporaries $\mathtt{t}_1$ and $\mathtt{t}_2$ in order to simplify expression evaluation and compressed multiple statements onto a single transition when they represent a single statement in the source program. □

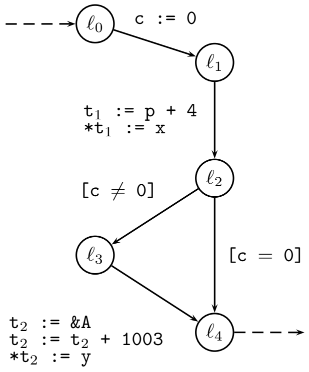
In order to reason about points-to and memory safety analyses, we need a memory model on which to base the concrete semantics. The unit of memory allocation is a *home* in the set $\mathbb{H}$. Each home h represents a contiguous block of memory cells, e.g., a statically declared array. A *location* $h[i]$ represents the cell at integer *offset* i in home h. The set of locations with homes from $\mathbb{H}$ is denoted $\mathbb{L}$. The function $\mathbf{size} : \mathbb{H} \rightarrow \mathbb{Z}^+$ maps a home to its allocated size. When

```
int A[4], c;

void bad(int *p, int x, int y) {
L0: c = 0;
L1: p[4] = x;
L2: if( c!=0 ) {
L3:     A[1003] = y;
L4: }
}

void ok(int *q, int n) {
L5: q[0] = n;
}

void main() {
    ok(A,0);
    bad(A,1,0);
}
```

(a)

$$\mathcal{V} = \{\mathtt{A}, \mathtt{c}, \mathtt{p}, \mathtt{x}, \mathtt{y}, \mathtt{t}_1, \mathtt{t}_2\}$$

$$\Gamma(v) = \begin{cases} 4, & \text{if } v = \mathtt{A} \\ 1, & \text{otherwise} \end{cases}$$



(b)

Figure 2.2: An unsafe C program

$0 \leq i < \mathbf{size}(h)$, location $h[i]$ is *in bounds*; otherwise it is *out of bounds*. Memory locations contain values from the set $\mathit{Vals} = \mathbb{Z} \cup \mathbb{L}$. A *memory state* is a partial function $m : \mathbb{L} \to \mathit{Vals}$. The set of all memory states is denoted $\mathbb{M}$. The set of concrete states C is the set of pairs (p, m) where $p \in \mathcal{L}$ represents the program position and m is a memory state.

An *allocation for* $\mathcal{V}$ is an injective function $\mathbf{home} : \mathcal{V} \to \mathbb{H}$ such that, for all $\mathtt{x} \in \mathcal{V}$, $\mathbf{size}(\mathbf{home}(\mathtt{x})) = \Gamma(\mathtt{x})$. Given such an allocation, the *lvalue* of $\mathtt{x} \in \mathcal{V}$ is $\mathbf{lval}(\mathtt{x}) = \mathbf{home}(\mathtt{x})[0]$. When the meaning is clear, we write `&x` for $\mathbf{lval}(\mathtt{x})$, $m(\mathtt{x})$ for $m(\mathbf{lval}(\mathtt{x}))$, and $m[\mathtt{x} \mapsto v]$ for $m[\mathbf{lval}(\mathtt{x}) \mapsto v]$, where m is a memory state. We say a location $h[i]$ is *within* a variable `x` if $h = \mathbf{home}(\mathtt{x})$ and $h[i]$ is in bounds.

Figure 2.3 defines the concrete interpretations $\mathcal{E}$ and **post** of, respectively, expressions and statements. Note that both $\mathcal{E}$ and **post** result in *sets* of, respectively, values and concrete states—the set-based semantics is needed as undefined operations may have a nondeterministic result. $\mathcal{E}$ returns the distinguished value $\perp$ in the case where an expression is not just ill-defined, but erroneous (e.g., reading an out-of-bounds memory location)—in this case the next state can have *any* memory state at *any* program point.

The operator $\widetilde{\oplus}$ is used to denote the semantic counterpart to the syntactic operator $\oplus$. The definition of $\widetilde{\oplus}$ is as usual for integer values. If an integer j is added to (alt. subtracted from) a location $h[i]$, where both $0 \leq i \leq \mathbf{size}(h)$ and $0 \leq i + j \leq \mathbf{size}(h)$ (alt. $0 \leq i - j \leq \mathbf{size}(h)$), then the result is $h[i + j]$ (alt. $h[i - j]$). If a location $h[j]$ is subtracted from a location $h'[i]$, where $h = h'$ and $0 \leq i, j \leq \mathbf{size}(h)$, the result is $i - j$. In all other cases, the result is undefined.

Note that arithmetic on pointer values is only defined for locations within (or one location beyond) a single home. E.g., adding an offset to a location sufficient to create an out-of-bounds location does not make the value point to a new home; subtracting locations from two different homes does not indicate the "distance" between the homes.

The operator $\widetilde{\trianglelefteq}$ is used to denote the semantic counterpart of a relational operator $\trianglelefteq$. The definition of $\widetilde{\trianglelefteq}$ is as usual for integer values. If two locations $h[i]$ and $h[j]$ have the same home, then $\widetilde{\trianglelefteq}$ is equal to the integer comparison $i \mathrel{\widetilde{\trianglelefteq}} j$. The value of $\widetilde{\trianglelefteq}$ is otherwise undefined, with the following exception: equality (resp. disequality) on two in-bounds locations with different homes or between an in-bounds location and 0 (the *null pointer constant*) evaluates to False (resp. True).

We now define the concrete interpretation of a program.

Definition 8. The *concrete semantics* $[\![\mathcal{P}]\!]_{\mathcal{C}}$ of a program $\mathcal{P} = (\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$

$$\mathcal{E}(m, n) = \{n\}$$

$$\mathcal{E}(m, \mathtt{x}) = \begin{cases} \mathbb{Z}, & \text{if } m(\mathtt{x}) \text{ is undefined} \\ \{m(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}(m, *\mathtt{x}) = \begin{cases} \bot, & \text{if } m(\mathtt{x}) \text{ is undefined, not a location, or out of bounds} \\ \mathbb{Z} & \text{if } m(m(\mathtt{x})) \text{ is undefined} \\ \{m(m(\mathtt{x}))\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}(m, \mathtt{x} \oplus \mathtt{y}) = \begin{cases} \mathbb{Z}, & \text{if } m(\mathtt{x}),\ m(\mathtt{y}), \text{ or } m(\mathtt{x}) \mathrel{\widetilde{\oplus}} m(\mathtt{y}) \text{ is undefined} \\ \{m(\mathtt{x}) \mathrel{\widetilde{\oplus}} m(\mathtt{y})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}(m, \&\mathtt{x}) = \{\mathbf{lval}(\mathtt{x})\}$$

$$\mathcal{E}(m, \mathtt{x} \trianglelefteq \mathtt{y}) = \begin{cases} \{0, 1\}, & \text{if } m(\mathtt{x}),\ m(\mathtt{y}), \text{ or } m(\mathtt{x}) \mathrel{\widetilde{\trianglelefteq}} m(\mathtt{y}) \text{ is undefined} \\ \{m(\mathtt{x}) \mathrel{\widetilde{\trianglelefteq}} m(\mathtt{y})\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}(m, p, \mathtt{x} \mathrel{\mathtt{:=}} E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } \mathcal{E}(m, E) = \bot \\ \{(p, m[\mathtt{x} \mapsto v]) \mid v \in \mathcal{E}(m, E)\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}(m, p, *\mathtt{x} \mathrel{\mathtt{:=}} E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } m(\mathtt{x}) \text{ is undefined, not a location, or out of bounds; or if } \mathcal{E}(m, E) = \bot \\ \{(p, m[m(\mathtt{x}) \mapsto v]) \mid v \in \mathcal{E}(m, E)\}, & \text{otherwise.} \end{cases}$$

$$\mathbf{post}(m, p, \mathtt{[}E\mathtt{]}) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } \mathcal{E}(m, E) = \bot \\ \emptyset, & \text{if } \mathcal{E}(m, E) = \{0\} \\ \{(p, m)\}, & \text{otherwise.} \end{cases}$$

Figure 2.3: The concrete interpretation.

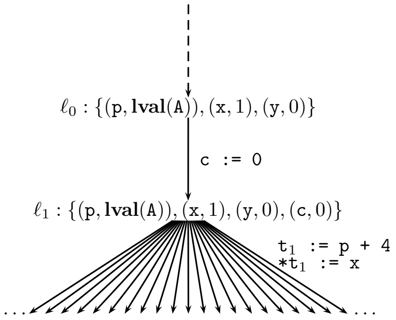


Figure 2.4: Concrete semantics for the program in Fig. 2.2(b)

is defined by the analysis $\mathcal{C} = (D_\mathcal{C}, \mathcal{I}_\mathcal{C}, \mathcal{F}_\mathcal{C})$, where

$$\mathcal{I}_\mathcal{C}[\mathcal{P}] = \{(en, m) \mid \forall l \in \mathbb{L}.\ m(l) \text{ is not a location}\}$$

$$\mathcal{F}_\mathcal{C}[\mathcal{P}](p, m) = \bigcup_{(p,S,p') \in \tau} \mathbf{post}(m, p', S)$$

If (p, S, p') is in τ and $c' \in \mathbf{post}(m, p', S)$, we say c' is an *S-successor* of (p, m).

Example 2.2. Figure 2.4 gives a subset of the reachable concrete states of the program in Fig. 2.2(b). At ℓ_0, `p` is **lval**`A` (the base address of the array `A`), `x` is 1, and `y` is 0. At ℓ_1, due to the assignment to out-of-bounds location `A`[4], the next state is undefined: *every* program point is reachable with *any* memory state. □

Note 2.1. The following modifications would allow us to model other aspects of the C programming language. To model dynamic memory allocation, we could add a component to the memory state to map locations to their allocation status. To model fixed-size integers and narrowing casts, we would need to define scalar values that span multiple locations (i.e., bytes), which require the modeling of alignment, padding, and byte order. To model functions, we could add an explicit representation of the call stack and variable scope. Handling strict aliasing would require more detailed type information, and would introduce more undefined behaviors in the case where a location is accessed through an illegal "type pun". This would correspondingly weaken the soundness results below, by replacing SafeDeref with a stronger predicate that captures the undefined behaviors introduced by disallowed casts. □

2.2 The Points-To Abstraction

The goal of pointer analysis is to compute an over-approximate points-to set for each variable in the program, i.e., the set of homes "into" which a variable may point in some reachable state.

A *points-to state* is a relation between variables. We denote the set of points-to states by Pts. When it is convenient, we treat a points-to state also as a relation between variables and memory locations: for points-to state pts, variables $\mathtt{x}, \mathtt{y}$, and location $h[i]$, we say $(\mathtt{x}, h[i])$ is in pts when $(\mathtt{x}, \mathtt{y})$ is in pts and $h[i]$ is within $\mathtt{y}$ (i.e., $h[i]$ is in bounds and $h = \mathbf{home}(\mathtt{y})$). We write $\mathit{pts}(\mathtt{x})$ for the *points-to set* of the variable $\mathtt{x}$ in pts, i.e., the set of variables $\mathtt{y}$ (alt. locations l) such that $(\mathtt{x}, \mathtt{y})$ (alt. $(\mathtt{x}, l)$) is in pts.

The concretization function γ_{Pts} takes a points-to state to the set of concrete states where at *most* its points-to relationships hold. Say that variable $\mathtt{x}$ *points to* $\mathtt{y}$ in memory state m if there exist locations l_1, l_2 such that l_1 is within $\mathtt{x}$, l_2 is within $\mathtt{y}$, and $m(l_1) = l_2$. Then m is in $\gamma_{\mathit{Pts}}(\mathit{pts})$ iff for all $\mathtt{x}, \mathtt{y}$ such that $\mathtt{x}$ points to $\mathtt{y}$ in m, the pair $(\mathtt{x}, \mathtt{y})$ is in pts. Note that there may be other pairs in pts as well—the points-to relation is over-approximate. Note also that *only in-bounds location values must agree with the points-to state*; out-of-bounds locations are unconstrained.

Note 2.2. Since the set of variables is known *a priori* and fixed, we will assume for convenience that the function **home** is a bijection. To extend our results to programs with dynamic allocation, we would have to introduce an inverse map from homes to abstract variables, such that every allocated block of memory has some representative in the domain of the points-to relation. For example, we could introduce a set of variables $\mathtt{malloc}_p$, for p in $\mathcal{L}$, such that any memory location allocated by a call to `malloc` at program point p is within $\mathtt{malloc}_p$. In any case, we would like to maintain the property: if $m(\mathtt{x})$ is an in-bounds location, then $m(\mathtt{x})$ is within some (perhaps abstract) variable `y`. □

Figure 2.5 defines the interpretations $\mathcal{E}_{Pts}$ and $\mathbf{post}_{Pts}$ for, respectively, expressions and statements in the points-to domain. The interpretations are chosen to match those used by common points-to analyses. A key feature is the treatment of the indirection operator `*`, which assumes that its argument is within bounds. Without this assumption, the interpretation would have to use the "top" points-to state (i.e., all pairs of variables) for the result of any indirect assignment.

We lift *Pts* to the set $\mathcal{L} \times Pts$ in the natural way.

Definition 9. A *flow- and path-sensitive points-to analysis* **Pts** is given by the tuple $(Pts, \mathcal{I}_{Pts}, \mathcal{F}_{Pts})$, where

$$\mathcal{I}_{Pts}[\mathcal{P}] = \{(en, \emptyset)\}$$

$$\mathcal{F}_{Pts}[\mathcal{P}](p, pts) = \bigcup_{(p,S,p') \in \tau} (p', \mathbf{post}_{Pts}(pts, S))$$

Example 2.3. Figure 2.6 shows a subset of the reachable points-to states for the program in Fig. 2.2(b). At ℓ_0, `p` points to `A`. The transition from ℓ_1 to ℓ_2 causes $\mathtt{t_1}$ to point to `A` as well. The presence of an out-of-bounds array access has no effect

$$\begin{aligned}
\mathcal{E}_{Pts}(pts, n) &= \emptyset \\
\mathcal{E}_{Pts}(pts, \mathtt{x}) &= pts(\mathtt{x}) \\
\mathcal{E}_{Pts}(pts, *\mathtt{x}) &= \{\mathtt{z} \in \mathcal{V} \mid \exists \mathtt{y} \in \mathcal{V} : pts(\mathtt{x}, \mathtt{y}) \land pts(\mathtt{y}, \mathtt{z})\} \\
\mathcal{E}_{Pts}(pts, \mathtt{x} \oplus \mathtt{y}) &= pts(\mathtt{x}) \cup pts(\mathtt{y}) \\
\mathcal{E}_{Pts}(pts, \&\mathtt{x}) &= \{\mathtt{x}\} \\
\mathcal{E}_{Pts}(pts, \mathtt{x} \trianglelefteq \mathtt{y}) &= \emptyset
\end{aligned}$$

$$\begin{aligned}
\mathbf{post}_{Pts}(pts, \mathtt{x} \ \mathtt{:=}\ E) &= pts \cup \{(\mathtt{x}, \mathtt{y}) \mid \mathtt{y} \in \mathcal{E}_{Pts}(pts, E)\} \\
\mathbf{post}_{Pts}(pts, *\mathtt{x} \ \mathtt{:=}\ E) &= \bigcup_{(\mathtt{x},\mathtt{y}) \in pts} \mathbf{post}_{Pts}(pts, \mathtt{y} \ \mathtt{:=}\ E) \\
\mathbf{post}_{Pts}(pts, \mathtt{[}E\mathtt{]}) &= pts
\end{aligned}$$

Figure 2.5: Abstract interpretation over points-to states.

on the points-to state: the analysis assumes that evaluating $*\mathtt{t}_1$ is safe. □

Definition 10. Let Deref be the predicate on $C \times \mathcal{V}$ that holds for concrete state (p, m) and variable `x` if, for some transition (p, S, p') in τ, S includes an expression of the form `*x`. Let SafeDeref be the predicate that holds in a concrete state (p, m) if, for all variables `x` such that Deref$((p, m), \mathtt{x})$ holds, $m(\mathtt{x})$ is an in-bounds location.

To show that **Pts** is SafeDeref-sound, we must first show that the function $\mathcal{E}_{Pts}$ over-approximates the locations given by the function $\mathcal{E}$ on concrete states in SafeDeref. We formalize this with the following Lemma.

Lemma 2.1. *For concrete state $c = (p, m)$, points-to state pts, variable `x`, location l, and expression E: if (1) SafeDeref(c) holds, (2) m is in $\gamma_{Pts}(pts)$, (3) there exists an edge (p, S, p') in τ such that E appears in S, (4) l is in $\mathcal{E}(m, E)$, and (5) l is within `x`, then `x` is in $\mathcal{E}_{Pts}(pts, E)$.*

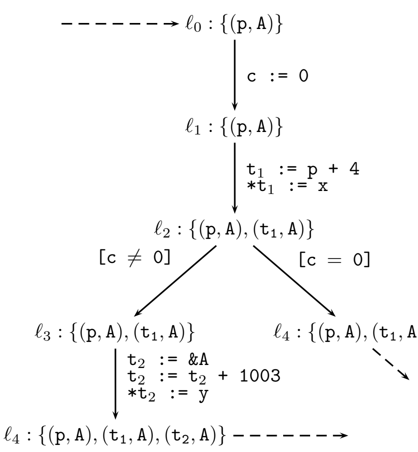


Figure 2.6: Points-to semantics for the program in Fig. 2.2(b)

Proof. We proceed by cases over E.

- $E = n$. No l can satisfy (4). The property holds trivially.

- $E = \mathtt{y}$. If $m(\mathtt{y})$ is undefined, then $\mathcal{E}(m, E) = \mathbb{Z}$ and no l can satisfy (4). Assume $m(\mathtt{y})$ is defined. Then, by definition, the only value in $\mathcal{E}(m, E)$ is $m(\mathtt{y})$. Assume $m(\mathtt{y})$ is a location within $\mathtt{x}$. Since m is in $\gamma_{Pts}(pts)$, $(\mathtt{y}, \mathtt{x})$ must be in pts. Hence, $\mathtt{x}$ is in $\mathcal{E}_{Pts}(pts, E)$.

- $E = *\mathtt{y}$. Since c is a SafeDeref state, $m(\mathtt{y})$ must be an in-bounds location l'. If $m(l')$ is undefined, then $\mathcal{E}(m, E) = \mathbb{Z}$ and no l can satisfy (4). Assume $m(l')$ is defined. Then, by definition, the only value in $\mathcal{E}(m, E)$ is $m(l')$. Assume $m(l')$ is a location within $\mathtt{x}$. Take $\mathtt{z}$ such that l' is within $\mathtt{z}$: such a $\mathtt{z}$ is guaranteed to exist if **home** is a bijection (see Note 2.2). Since m is in $\gamma_{Pts}(pts)$, $(\mathtt{y}, \mathtt{z})$ and $(\mathtt{z}, \mathtt{x})$ must be in pts. Hence, $\mathtt{x}$ is in $\mathcal{E}_{Pts}(pts, E)$.

- $E = \mathtt{y} \oplus \mathtt{z}$. The only cases where $\mathcal{E}(m, E)$ contains a location value is when $\oplus$ is `+` or `-`, one of the operands is location, the other is an integer, and the result is well-defined. We will consider only the case for `+`; the case for `-` is similar.

Assume, wlog, that $m(\mathtt{y}) = h[i]$ (with $0 \leq i \leq \mathbf{size}(h)$) and $m(\mathtt{z}) = j$ (with $0 \leq i + j \leq \mathbf{size}(h)$). By definition, the only value in $\mathcal{E}(m, E)$ is $h[i + j]$. Assume $h[i + j]$ is within $\mathtt{x}$. Then $h[i]$ is within $\mathtt{x}$. Since m is in $\gamma_{Pts}(pts)$, $\mathtt{x}$ must be in $pts(\mathtt{y})$. Hence, $\mathtt{x}$ is in $\mathcal{E}_{Pts}(pts, E)$.

- $E = \&\mathtt{x}$. By definition, the only value in $\mathcal{E}(m, E)$ is $\mathbf{lval}(\mathtt{x})$, which is within $\mathtt{x}$. By definition, $\mathtt{x}$ is in $\mathcal{E}_{Pts}(pts, E)$.
- $E = \mathtt{y} \trianglelefteq \mathtt{z}$. No l can satisfy (4). The property holds trivially.

$\square$

Theorem 2.2. *The points-to analysis* **Pts** *is* SafeDeref*-sound.*

Proof. Applying Theorem 1.4, it suffices to show that $\mathcal{I}_{Pts}$ is sound and $\mathcal{F}_{Pts}$ is SafeDeref-sound.

Take a program $\mathcal{P} = (\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$. We must show: (1) $\mathcal{I}_{Pts}[\mathcal{P}]$ over-approximates $\mathcal{I}_{\mathcal{C}}[\mathcal{P}]$ and (2) $\mathcal{F}_{Pts}[\mathcal{P}](p, pts)$ over-approximates $\mathcal{F}_{\mathcal{C}}[\mathcal{P}](p, m)$ whenever (p, m) is a SafeDeref state and m is in $\gamma_{Pts}(pts)$.

(1) Let (en, m) be an initial concrete state. By the definition of $\mathcal{I}_{\mathcal{C}}$, $m(l)$ is not a location for any $l \in \mathbb{L}$. By the definition of $\mathcal{I}_{Pts}$, the only initial points-to state is $(en, \emptyset)$. By the definition of γ_{Pts}, m is in $\gamma_{Pts}(\emptyset)$. Hence, (en, m) is in $\gamma_{Pts}(\mathcal{I}_{Pts}[\mathcal{P}])$. This shows soundness for $\mathcal{I}_{Pts}$.

(2) Take concrete states $c = (p, m)$, c', points-to state pts, and statement S such that m is in $\gamma_{Pts}(pts)$, SafeDeref(c) holds, and $c' = (p', m')$ is an S-successor of c. Note that $\mathcal{E}(m, E)$ cannot be $\perp$, since c is a SafeDeref state. It suffices to show that c' is in $\gamma_{Pts}(\mathcal{F}_{Pts}[\mathcal{P}](p, pts))$. We proceed by cases on S:

- $S = \texttt{x := } E$. By definition, $m' = m[\texttt{x} \mapsto v]$ for some value v in $\mathcal{E}(m, E)$. The only interesting case is when v is an in-bounds location l within a variable $\texttt{y}$. It suffices to show $\texttt{y}$ is in $\mathcal{E}_{Pts}(pts, E)$. This follows from Lemma 2.1.

- $S = \texttt{*x := } E$. Since c is a SafeDeref state, $m(\texttt{x})$ must be an in-bounds location l. Hence, $m' = m[l \mapsto v]$ for some value v in $\mathcal{E}(m, E)$. Since m is in $\gamma_{Pts}(pts)$, if l is within variable $\texttt{y}$, we must have $pts(\texttt{x}, \texttt{y})$. The remainder of the proof is as in the previous case, with $S = \texttt{y := } E$.

- $S = \texttt{[}E\texttt{]}$. Since c is a SafeDeref state, $\mathcal{E}(m, E)$ cannot be $\perp$. Thus, the only interesting case is where $c = c'$. By definition, $\mathbf{post}_{Pts}(pts, S)$ is equal to pts. Hence, c' is in $\gamma_{Pts}(\mathbf{post}_{Pts}(pts, S))$.

This shows SafeDeref-soundness for $\mathcal{F}_{Pts}$.

Hence, **Pts** is SafeDeref-sound. □

We can extract more traditional flow-sensitive, global, and flow-insensitive pointer analyses from $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ as follows.

- A *flow-sensitive, program-point-sensitive (path-insensitive) analysis* is derived by assigning to each program point p the least points-to state (by subset inclusion) $pts^\sharp$ such that, if (p, pts) is in $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$, then $pts \subseteq pts^\sharp$.

- A *flow-sensitive, global (program-point-insensitive) analysis* is derived by assigning to every program point the least points-to state (by subset inclusion) $pts^\sharp$ such that, if (p, pts) is in $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ for *any* program point p, then $pts \subseteq pts^\sharp$.

- A *flow-insensitive analysis* is derived by replacing τ in Definition 9 with the relation $\tau^\sharp$, where the edge (p, S, q) is in $\tau^\sharp$ whenever some edge (t, S, u) is in τ, for *any* program points t and u. Intuitively, if a statement occurs anywhere in the program, then it may occur between any two program points—the interpretation ignores the control-flow structure of the program.

- *Flow-insensitive, program-point-sensitive* and *flow-insensitive, global* combinations can be defined as above, substituting the flow-insensitive semantics for $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$.

Theorem 2.3. *Each of the flow-, path-, and program-point-sensitive and insensitive variations of the points-to analysis is* SafeDeref*-sound.*

Proof. Take a program $\mathcal{P}$ and a points-to analysis $\mathcal{Q}$ from among those described above. It is clear that, for any state (p, pts) in $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$, there is a state (p, pts') in $[\![\mathcal{P}]\!]_{\mathcal{Q}}$ such that $pts \subseteq pts'$. Since γ_{Pts} is monotonic, any concrete state in $\gamma_{Pts}([\![\mathcal{P}]\!]_{\mathbf{Pts}})$ is also in $\gamma_{Pts}([\![\mathcal{P}]\!]_{\mathcal{Q}})$. By Theorem 2.2, $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ over-approximates $[\![\mathcal{P}]\!]_{\mathcal{C}\downarrow\textsc{SafeDeref}}$. Hence, $[\![\mathcal{P}]\!]_{\mathcal{Q}}$ over-approximates $[\![\mathcal{P}]\!]_{\mathcal{C}\downarrow\textsc{SafeDeref}}$: $\mathcal{Q}$ is SafeDeref-sound. □

Note 2.3. The flow-sensitive, program-point-sensitive analysis yields a points-to relation similar to that of Emami et al. [24]. The flow-insensitive, global analysis procedure yields a points-to relation similar to that of Andersen [3]. The Steensgaard [63] and Das [18] relations add additional approximation to the global relation. We claim (but do not prove formally here) that these procedures approximate $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ and, thus, are *at least* SafeDeref-sound. □

In summary, we have shown that a set of points-to analyses which share the assumptions of widely used analyses from the literature are sound for all memory-safe executions. This claim is both stronger and more precise than any correctness claims we have encountered: our points-to analyses (and, by extension, those cited above) compute a relation which is conservative not only for "well-behaved" (i.e., memory-safe) programs, but for all well-behaved *executions*, even the well-behaved executions of ill-behaved programs.

By the definition of conditional soundness, it is possible some condition θ weaker than SafeDeref exists such that some or all of the above analyses are θ-sound. We will show that this is not the case: no "reasonable" points-to analysis is θ-sound for any θ weaker than SafeDeref.

We have shown that, if we can prove the absence of non-SafeDeref states in $[\![\mathcal{P}]\!]_{\mathcal{C}}$, the points-to analyses we have defined above will be sound. It remains to describe an analysis parameterized by points-to information which can perform a precise memory safety analysis.

2.3 Optimality of SafeDeref-Soundness

We will show that SafeDeref-soundness is the best we can hope for from any "reasonable" analysis on the points-to domain. For our purposes a "reasonable" analysis is one that:

1. Is θ-sound for some computable θ,
2. Has a θ-sound transfer function, and
3. Does not produce trivial results for large classes of non-pathological programs.

The first requirement prevents us from defining θ to be, for example, "the set of SafeDeref states plus all non-SafeDeref states that are not reachable in any computation." This predicate is clearly weaker than SafeDeref and would allow us to define a conditionally sound analysis—however, it would require reference to a reachability predicate which is only semidecidable [65].

The second requirement means, in essence, that the argument for θ-soundness must be inductive. While it is possible to imagine an analysis which somehow achieves θ-soundness in spite of a non-θ-sound transfer function, such an analysis would be decidedly odd. The standard practice of the static analysis community is to build sound analyses out of sound components, using syntax-directed inductive reasoning.

The final requirement means the analysis can't "cheat" by simply giving up on programs for which a precise result can be computed. To be more precise, we say a θ-sound analysis $\mathcal{A}$ is *trivial for program* $\mathcal{P}$ if there exists some set of states $D' \subseteq D_{\mathcal{A}}$ such that $[\![\mathcal{P}]\!]_{C}{\downarrow_\theta} \subseteq \gamma_{\mathcal{A}}(D') \neq C$, but $\gamma_{\mathcal{A}}([\![\mathcal{P}]\!]_{\mathcal{A}}) = C$.

Any analysis might be trivial for certain pathological programs. For example, **Pts** is trivial for the program

```
while(1) { x = &x; x = &y; y = x; }
```

which will have the points-to relation $\{(\mathtt{x},\mathtt{x}),(\mathtt{x},\mathtt{y}),(\mathtt{y},\mathtt{x}),(\mathtt{y},\mathtt{y})\}$ at every program point even though `y` will never point to `x` in any concrete execution. What we will attempt to show below is that a non-SafeDeref-sound points-to analysis will necessarily be trivial for a broad class of non-pathological programs.

To characterize those programs more precisely, we will define a set of states that are essentially indistinguishable in the points-to domain. First, we note the following important property of the points-to abstraction. (Note: we use $\mathit{Vals}_{\bot}$ to denote the set *Vals* augmented with the distinguished "undefined" value $\bot$; i.e., $m[\mathtt{x} \mapsto \bot]$ when `x` is undefined in m.)

Lemma 2.4. *Let (p, m) be a concrete state and* $\texttt{x}$ *a variable such that $m(\texttt{x})$ is undefined, not a location, or out of bounds. Let (p, m') be a concrete state such that $m' = m[\texttt{x} \mapsto v]$, for some $v \in \mathit{Vals}_\perp$. If points-to state (p, pts) over-approximates (p, m'), then it also over-approximates (p, m).*

Proof. By the definition of γ_{Pts}, *pts* over-approximates m iff for all $\texttt{y}, \texttt{z}$ such that $\texttt{y}$ points to $\texttt{z}$ in m, $(\texttt{y}, \texttt{z})$ is in *pts*. Since m and m' differ only at $\texttt{x}$, this holds for all $\texttt{y}$ distinct from $\texttt{x}$. Since $m(\texttt{x})$ is undefined, not a location, or out of bounds, it can't point to any location. Hence, *pts* over-approximates m. □

If a program has a transition (p, S, p') where S includes an expression of the form $\texttt{*x}$, we call $\texttt{x}$ *a* SafeDeref *trigger at program-point p*, denoted $\texttt{x}_p$. The function ρ maps a set of concrete states C' to a set of states which are SafeDeref-*equivalent to C'*. The set $\rho(C')$ includes all of the states in C' that are also SafeDeref states and, for every non-SafeDeref state (p, m) in C', all of the states (p, m') that differ from (p, m) only at a SafeDeref trigger:

$$\rho(C') = \{(p, m[\texttt{x}_p \mapsto v]) \mid (p, m) \in (C' - \textsc{SafeDeref}), v \in \mathit{Vals}_\perp\} \cup (C' \cap \textsc{SafeDeref})$$

The function ρ allows us to precisely characterize the problematic programs for a θ-sound, non-SafeDeref-sound analysis: they are exactly the programs that reach a state in $\rho(\theta - \textsc{SafeDeref})$. If θ includes a non-SafeDeref concrete state (p, m), then such an analysis will be trivial for any program $\mathcal{P}$ that reaches p in any state m'—even a SafeDeref state—differing from m only at $\texttt{x}_p$.

Theorem 2.5. *Let $\mathcal{Q}$ be an analysis over the domain Pts with a θ-sound semantic interpretation, for some $\theta \not\subseteq \textsc{SafeDeref}$. Let $\mathcal{P}$ be a program such that:*

1. *There exists some* $P \subseteq \mathit{Pts}$ *such that* $[\![\mathcal{P}]\!]_{\mathcal{C}} \subseteq \gamma_{\mathit{Pts}}(P) \neq C$*, and*

2. $[\![\mathcal{P}]\!]_{\mathcal{C}}{\downarrow_\theta}$ *contains at least one state in* $\rho(\theta - \textsc{SafeDeref})$*.*

$\mathcal{Q}$ *is trivial for* $\mathcal{P}$*.*

Proof. Let (p, m) be a state in $[\![\mathcal{P}]\!]_{\mathcal{C}}{\downarrow_\theta} \cap \rho(\theta - \textsc{SafeDeref})$. Let (p, pts) be a state in $[\![\mathcal{P}]\!]_{\mathcal{Q}}$ that over-approximates (p, m). By the definition of ρ, there is some state (p, m') in $(\theta - \textsc{SafeDeref})$ such that $m = m'[\mathtt{x}_p \mapsto v]$ for some $v \in \mathit{Vals}_\perp$. Since (p, m') is not a $\textsc{SafeDeref}$-state, $m'(\mathtt{x})$ must be undefined, not a location, or out of bounds. By Lemma 2.4, (p, pts) also over-approximates (p, m'). Hence, (p, m') is in $\gamma_{\mathit{Pts}}(p, \mathit{pts}) \cap \theta$. Since $\mathcal{F}_{\mathcal{Q}}$ is θ-sound, $\mathcal{F}_{\mathcal{Q}}[\mathcal{P}](p, \mathit{pts})$ must over-approximate $\mathcal{F}_{\mathcal{C}}[\mathcal{P}](p, m')$. By definition, $\mathcal{F}_{\mathcal{C}}[\mathcal{P}](p, m') = C$. Hence, $\gamma_{\mathit{Pts}}(\mathcal{F}_{\mathcal{Q}}[\mathcal{P}](p, \mathit{pts})) = C$ and, by monotonicity, $\gamma_{\mathit{Pts}}([\![\mathcal{P}]\!]_{\mathcal{Q}}) = C$. But, by assumption, there exists some $P \subseteq \mathit{Pts}$ such that $[\![\mathcal{P}]\!]_{\mathcal{C}} \subseteq \gamma_{\mathit{Pts}}(P) \neq C$. Hence, $\mathcal{Q}$ is trivial for $\mathcal{P}$. $\square$

Theorem 2.5 makes a rather modest claim, and we have already discussed several of its limiting assumptions, but we would like to be the first to point out two more obvious limitations.

First, it depends strongly on the definition of γ_{Pts} given in Section 2.2. It is possible that a different concretization function could yield a tighter soundness result. It is our belief no such concretization function exists.

Second, the Theorem becomes vacuous if there is no program meeting conditions (1) and (2). Indeed, we can ensure this is the case by choosing θ to be the set of concrete states such that any program $\mathcal{P}$ reaching a state in $\rho(\theta - \textsc{SafeDeref})$ has $[\![\mathcal{P}]\!]_{\mathcal{C}} = C$. We claim this possibility is ruled out by the assumption that θ is practically computable. In practice, we expect there to be many programs satisfying the conditions of the Theorem for any realistic θ.

2.4 Checking Memory Safety

We wish to define an analysis procedure that will soundly prove the absence of non-SafeDeref states in the concrete program. Note that the only attributes of a location value that are relevant to the property SafeDeref are its offset and the size of its home; if we can precisely track these attributes, we can ignore the home component of a location (i.e., which variable it is within) so long as we have access to over-approximate points-to information.

Note 2.4. In our description of the analysis, we will omit the merging, widening, and covering operations necessary to make the reachability computation tractable. □

Our analysis will track *abstract values* from the set $\widehat{Vals}$. An abstract value is either an integer or an *abstract location*, a pair (i, n) representing a location at offset i in a home of size n. Each abstract value $\hat{v}$ represents a set of concrete values, according to the abstraction function $\alpha : Vals \rightarrow \widehat{Vals}$. For integer values, α is the identity (i.e., $\alpha(n) = n$); for concrete location values, α preserves the offset and size (i.e., $\alpha(h[i]) = (i, \mathbf{size}(h))$). An abstract location (i, n) is *in bounds* if it represents only in bounds concrete locations (i.e., $0 \leq i < n$); otherwise it is *out of bounds*. An *abstract memory state* is a partial function $b : \mathbb{L} \rightarrow \widehat{Vals}$. We denote by B the set of abstract memory states.

The concretization function $\gamma_B : B \rightarrow 2^C$ takes an abstract memory state b to the set of concrete memories abstracted by b. A concrete memory m is in $\gamma_B(b)$ iff for all l either $m(l)$ and $b(l)$ are both undefined or $\alpha(m(l)) = b(l)$.

Figure 2.7 defines the interpretations $\mathcal{E}_B$ and $\mathbf{post}_B$ for, respectively, expressions and statements with respect to B. Note that the interpretations rely on points-to information. In the limiting case, where no points-to information is available (i.e., the points-to relation includes all pairs), the expression `*x` can take the value of any location abstracted by $b(\mathtt{x})$. As in the concrete interpretation, $\mathcal{E}_B$ returns the value $\bot$ in the case where expression evaluation is (potentially) erroneous.

The operator $\widehat{\oplus}$ is used to denote the abstract counterpart to the syntactic operator $\oplus$. The definition of $\widehat{\oplus}$ is as usual for integer values. If an integer j is added to (resp. subtracted from) an abstract location (i, m), where both $0 \leq i \leq m$ and $0 \leq i + j \leq m$ (resp. $0 \leq i - j \leq m$), the result is $(i + j, m)$ (resp. $(i - j, m)$). In all other cases, the result is undefined.

The operator $\widehat{\trianglelefteq}_A$ is used to denote the abstract counterpart to the syntactic operator $\trianglelefteq$, parameterized by a points-to set $A \subseteq \mathcal{V}$. The definition of $\widehat{\trianglelefteq}_A$ is as usual for integer values. If two in-bounds abstract location values (i, m) and (j, n) are compared for equality (resp. disequality) and either $i \neq j$, $m \neq n$, or $A = \emptyset$, then the result is 0 (resp. 1). In all other cases, $\widehat{\trianglelefteq}_A$ is undefined.

We lift B to the domain $\mathcal{L} \times B$ in the natural way.

Definition 11. The analysis generator $\widetilde{\mathcal{B}}$ maps a set of states $Q \subseteq \mathcal{L} \times \mathit{Pts}$ to the memory safety analysis $\widetilde{\mathcal{B}}\langle Q\rangle$ defined by the parameterized interpretations

$$\widetilde{\mathcal{I}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}] = \{(\mathit{en}, b) \mid \forall l \in \mathbb{L} : b(l) \text{ is not a location }\}$$
$$\widetilde{\mathcal{F}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}](p, b) = \bigcup_{(p,S,p')\in\tau} \bigcup_{(p,\mathit{pts})\in Q} \mathbf{post}_B(b, \mathit{pts}, p', S)$$

Lemma 2.6. *For concrete memory state m, points-to state pts, abstract memory state b, value v, and expression E: if (1) m is in $\gamma_{\mathit{Pts}}(\mathit{pts}) \cap \gamma_B(b)$ and (2) v is in $\mathcal{E}(m, E)$, then $\alpha(v) = \hat{v}$ for some $\hat{v}$ in $\mathcal{E}_B(b, \mathit{pts}, E)$.*

Proof. We proceed by cases on E.

$$\mathcal{E}_B(b, pts, n) = \{n\}$$

$$\mathcal{E}_B(b, pts, \texttt{x}) = \begin{cases} \mathbb{Z}, & \text{if } b(\texttt{x}) \text{ is undefined} \\ \{b(\texttt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_B(b, pts, \texttt{*x}) = \begin{cases} \bot, & \text{if } b(\texttt{x}) \text{ is undefined, not a location, or out of bounds} \\ \widehat{Vals}, & \text{if } b(l) \text{ is undefined for some } l \text{ in } pts(\texttt{x}) \text{, where } \alpha(l) = b(\texttt{x}) \\ \{b(l) \mid pts(\texttt{x}, l),\ \alpha(l) = b(\texttt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_B(b, pts, \texttt{x} \oplus \texttt{y}) = \begin{cases} \mathbb{Z}, & \text{if } b(\texttt{x}),\, b(\texttt{y}) \text{, or } b(\texttt{x}) \mathbin{\widehat{\oplus}} b(\texttt{y}) \text{ is undefined} \\ \{b(\texttt{x}) \mathbin{\widehat{\oplus}} b(\texttt{y})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_B(b, pts, \texttt{\&x}) = \{(0, \mathbf{size}(\mathbf{home}(\texttt{x})))\}$$

$$\mathcal{E}_B(b, pts, \texttt{x} \trianglelefteq \texttt{y}) = \begin{cases} \{0, 1\}, & \text{if } b(\texttt{x}) \mathbin{\widehat{\trianglelefteq}}_{pts(\texttt{x}) \cap pts(\texttt{y})} b(\texttt{y}) \text{ is undefined} \\ b(\texttt{x}) \mathbin{\widehat{\trianglelefteq}}_{pts(\texttt{x}) \cap pts(\texttt{y})} b(\texttt{y}) & \text{otherwise} \end{cases}$$

$$\mathbf{post}_B(b, pts, p, \texttt{x := } E) = \begin{cases} \mathcal{L} \times B, & \text{if } \mathcal{E}_B(b, pts, E) = \bot \\ \{(p, b[\texttt{x} \mapsto \hat{v}]) \mid \hat{v} \in \mathcal{E}_B(b, pts, E)], & \text{otherwise} \end{cases}$$

$$\mathbf{post}_B(b, pts, p, \texttt{*x := } E) = \begin{cases} \mathcal{L} \times B, & \text{if } b(\texttt{x}) \text{ is undefined, not a location, or out of bounds;} \\ & \text{or if } \mathcal{E}_B(b, pts, E) = \bot \\ \{(p, b[l \mapsto \hat{v}]) \mid pts(\texttt{x}, l),\ \alpha(l) = b(\texttt{x}),\ \hat{v} \in \mathcal{E}_B(b, pts, E)\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}_B(b, pts, p, \texttt{[}E\texttt{]}) = \begin{cases} \mathcal{L} \times B, & \text{if } \mathcal{E}_B(b, pts, E) = \bot \\ \emptyset, & \text{if } \mathcal{E}_B(b, pts, E) = \{0\} \\ \{(p, b)\}, & \text{otherwise.} \end{cases}$$

Figure 2.7: Abstract interpretation over B.

- $E = n$. By definition, the only value in $\mathcal{E}(m, E)$ or $\mathcal{E}_B(b, pts, E)$ is n and $\alpha(n) = n$.

- $E = \mathtt{x}$. Assume $b(\mathtt{x})$ is undefined. Since m is in $\gamma_B(b)$, $m(\mathtt{x})$ must also be undefined. By definition, $\mathcal{E}(m, E) = \mathcal{E}_B(b, pts, E) = \mathbb{Z}$.

  Now, assume $b(\mathtt{x})$ is defined. Since m is in $\gamma_B(b)$, $m(\mathtt{x})$ must also be defined and $\alpha(m(\mathtt{x})) = b(\mathtt{x})$. By definition, the only value in $\mathcal{E}(m, E)$ is $m(\mathtt{x})$ and the only value in $\mathcal{E}_{Pts}(b, pts, E)$ is $b(\mathtt{x})$.

- $E = \mathtt{*x}$. Assume, wlog, that $m(\mathtt{x})$ is an in-bounds location l. Since m is in $\gamma_B(b)$, $b(\mathtt{x}) = \alpha(l)$. Since m is in $\gamma_{Pts}(pts)$, l must be in $pts(\mathtt{x})$.

  If $b(l')$ is undefined for some l' in $pts(\mathtt{x})$ such that $\alpha(l')$ is equal to $\alpha(l)$, then $\mathcal{E}_B(b, pts, E) = \widehat{Vals}$ and the property holds trivially. Assume $b(l')$ is defined for all such l'. In particular, $b(l)$ is defined. Since m is in $\gamma_B(b)$, $m(l)$ must also be defined and $\alpha(m(l)) = b(l)$. By definition, the only value in $\mathcal{E}(m, E)$ is $m(l)$ and, since l is in $pts(\mathtt{x})$, $b(l)$ is in $\mathcal{E}_B(b, pts, E)$.

- $E = \mathtt{x} \oplus \mathtt{y}$. If $b(\mathtt{x}) \mathbin{\widehat{\oplus}} b(\mathtt{y})$ is undefined, then $\mathcal{E}_{Pts}(b, pts, E) = \mathbb{Z}$. It suffices to show that there are no location values in $\mathcal{E}(m, E)$. The only case where $\mathcal{E}(m, E)$ contains a location value is when $\oplus$ is `+` or `-`, one of the operands is a location value, the other is an integer, and the result is well-defined. We will consider only the case for `+`; the case for `-` is similar. Assume, wlog, that $m(\mathtt{x})$ is an in-bounds location $h[i]$ and $m(\mathtt{y})$ is an integer j, with $0 \leq i+j \leq \mathbf{size}(h)$. By definition, the only value in $\mathcal{E}(m, E)$ is $h[i] \mathbin{\widehat{\oplus}} j = h[i+j]$. Since m is in $\gamma_B(b)$, $\alpha(m(\mathtt{x})) = b(\mathtt{x})$ and $\alpha(m(\mathtt{y})) = b(\mathtt{y})$. Hence, $b(\mathtt{x}) = (i, \mathbf{size}(h))$ and $b(\mathtt{y}) = j$. But $(i, \mathbf{size}(h)) \mathbin{\widehat{\oplus}} j$ is well-defined, which contradicts the assumption that $b(\mathtt{x}) \mathbin{\widehat{\oplus}} b(\mathtt{y})$ is undefined. Thus, there can be no location values in $\mathcal{E}(m, E)$.

Assume $b(\mathtt{x}) \mathbin{\widehat{\oplus}} b(\mathtt{y})$ is well-defined and both $b(\mathtt{x})$ and $b(\mathtt{y})$ are integers, say i and j. Since m is in $\gamma_B(b)$, $m(\mathtt{x})$ must be i and $m(y)$ must be j. By definition, the only values in $\mathcal{E}(m, E)$ and $\mathcal{E}_B(b, \mathit{pts}, E)$, respectively, are $i \mathbin{\widetilde{\oplus}} j$ and $i \mathbin{\widehat{\oplus}} j$. Since $\widetilde{\oplus}$ and $\widehat{\oplus}$ are defined in the same way for integer operands, $\alpha(i \mathbin{\widetilde{\oplus}} j) = i \mathbin{\widehat{\oplus}} j$.

Assume $b(\mathtt{x}) \mathbin{\widehat{\oplus}} b(\mathtt{y})$ is well-defined, one of $b(\mathtt{x})$, $b(\mathtt{y})$ is an abstract location (i, n) (with $0 \leq i \leq n$), the other an integer j (with $0 \leq i + j \leq n$), and $\oplus$ is `+`. Since m is in $\gamma_B(b)$, $m(\mathtt{x})$ must be an location $h[i]$ with $\mathbf{size}(h) = n$ and $m(\mathtt{y})$ must be j. By definition, the only values in $\mathcal{E}(m, E)$ and $\mathcal{E}_B(b, \mathit{pts}, E)$, respectively, are $h[i + j]$ and $(i + j, n)$, and $\alpha(h[i + j]) = (i + j, n)$.

The case where $\oplus$ is `-` and $b(\mathtt{x})$, $b(\mathtt{y})$ are abstract locations is similar.

- $E = \mathtt{\&x}$. By definition, the only values in $\mathcal{E}(m, E)$ and $\mathcal{E}_B(b, \mathit{pts}, E)$, respectively, are $\mathbf{lval}(\mathtt{x})$ and $(0, \mathbf{size}(\mathbf{home}(\mathtt{x})))$. By definition, $\alpha(\mathbf{lval}(\mathtt{x})) = \alpha(\mathbf{home}(\mathtt{x})[0]) = (0, \mathbf{size}(\mathbf{home}(\mathtt{x})))$.

- $E = \mathtt{x} \trianglelefteq \mathtt{y}$. If $b(\mathtt{x}) \mathbin{\widehat{\trianglelefteq}}_{\mathit{pts}(\mathtt{x}) \cap \mathit{pts}(\mathtt{y})} b(\mathtt{y})$ is undefined, then the property holds trivially.

  Assume $b(\mathtt{x}) \mathbin{\widehat{\trianglelefteq}}_{\mathit{pts}(\mathtt{x}) \cap \mathit{pts}(\mathtt{y})} b(\mathtt{y})$ is well-defined, and $b(\mathtt{x})$, $b(\mathtt{y})$ are both integers, say i and j. Since m is in $\gamma_B(b)$, $m(\mathtt{x})$ must be i and $m(y)$ must be j. By definition, the only values in $\mathcal{E}(m, E)$ and $\mathcal{E}_B(b, \mathit{pts}, E)$, respectively, are $i \mathbin{\widetilde{\trianglelefteq}} j$ and $i \mathbin{\widehat{\trianglelefteq}} j$. Since $\widetilde{\trianglelefteq}$ and $\widehat{\trianglelefteq}$ are defined in the same way for integer operands, $\alpha(i \mathbin{\widetilde{\trianglelefteq}} j) = i \mathbin{\widehat{\trianglelefteq}} j$.

  Assume $b(\mathtt{x}) \mathbin{\widehat{\trianglelefteq}}_{\mathit{pts}(\mathtt{x}) \cap \mathit{pts}(\mathtt{y})} b(\mathtt{y})$ is well-defined, $b(\mathtt{x})$ and $b(\mathtt{y})$ are in-bounds abstract locations, say (i, n) and (j, r), and $\trianglelefteq$ is `==`. Since m is in $\gamma_B(b)$, we have $m(\mathtt{x}) = h[i]$ (with $\mathbf{size}(h) = n$) and $m(\mathtt{y}) = h'[j]$, (with $\mathbf{size}(h') = r$). By definition, the only value in $\mathcal{E}_B(b, \mathit{pts}, E)$ is 0 and: $i \neq j$, $n \neq r$, or $\mathit{pts}(\mathtt{x}) \cap \mathit{pts}(\mathtt{y})$ is empty. We must show that the only value in $\mathcal{E}(m, E)$ is 0: this will be the case when $h[i] \neq h'[j]$. If $i \neq j$, this is immediate. If $n \neq r$, then $\mathbf{size}(h) \neq \mathbf{size}(h')$ and, thus, $h \neq h'$. If $\mathit{pts}(\mathtt{x}) \cap \mathit{pts}(\mathtt{y})$ is empty, then, since m is in $\gamma_{\mathit{Pts}}(\mathit{pts})$, $h \neq h'$.

The case when $\widehat{\trianglelefteq}$ is `!=` and $b(\mathtt{x})$, $b(\mathtt{y})$ are abstract locations is similar.

□

Lemma 2.7. *Let E be an expression, m a concrete memory state, and b an abstract memory state such that m is in $\gamma_B(b)$. $\mathcal{E}(m, E) = \bot$ iff $\mathcal{E}_B(b, \mathit{pts}, E) = \bot$, for all points-to states pts, .*

Proof. Assume, wlog, that E is of the form `*x`. $\mathcal{E}(m, E) = \bot$ iff $m(\mathtt{x})$ undefined, not a location, or out of bounds. Similarly, $\mathcal{E}_B(b, \mathit{pts}, E) = \bot$ iff $b(\mathtt{x})$ is undefined, not a location, or out of bounds, for all points-to states pts. Since m is in $\gamma_B(b)$, $m(\mathtt{x})$ is undefined, not a location, or out of bounds iff $b(\mathtt{x})$ is undefined, not a location, or out of bounds. □

Lemma 2.8. *$\widetilde{\mathcal{I}}_{\mathcal{B}}\langle Q\rangle$ is sound for every set of states Q.*

Proof. Let $\mathcal{P}$ be a program. We must show $\widetilde{\mathcal{I}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}]$ over-approximates $\mathcal{I}_{\mathcal{C}}[\mathcal{P}]$. Let $c = (p, m)$ be a state in $\mathcal{I}_{\mathcal{C}}[\mathcal{P}]$ and let $a = (p, b)$ be an abstract state such that c is in $\gamma_B(a)$. By definition, $m(l)$ is not a location for any l in $\boldsymbol{\mathcal{L}}$. Since, $b(l) = \alpha(m(l))$ whenever $b(l)$ is defined, $b(l)$ is also not a location for any l in $\boldsymbol{\mathcal{L}}$. Hence, a is in $\widetilde{\mathcal{I}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}]$. □

Lemma 2.9. *$\widetilde{\mathcal{F}}_{\mathcal{B}}\langle Q\rangle$ is $\gamma_{\mathit{Pts}}(Q)$-sound for every set of states Q.*

Proof. Let $\mathcal{P}$ be a program. We must show $\widetilde{\mathcal{F}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}](p, b)$ over-approximates $\mathcal{F}_{\mathcal{C}}[\mathcal{P}](p, m)$ whenever (p, m) is in $\gamma_{Pts}(Q)$ and m is in $\gamma_B(b)$.

Take concrete states $c = (p, m)$, $c' = (p', m')$, abstract state $a = (p, b)$, points-to state (p, pts), and statement S such that m is in both $\gamma_B(b)$ and $\gamma_{Pts}(pts)$ and c' is an S-successor of c. It suffices to show that there is some (p', b') in $\mathbf{post}_B(b, pts, p', S)$ such that m' is in $\gamma_B(b')$. We proceed by cases on S:

- $S = \texttt{x := } E$. If $\mathcal{E}(m, E) = \bot$, then $\mathcal{E}_B(b, pts, E) = \bot$, by Lemma 2.7, and the claim is trivial. Assume $\mathcal{E}(m, E) \neq \bot$. Then, $m' = m[\texttt{x} \mapsto v]$ for some value v in $\mathcal{E}(m, E)$. By Lemma 2.6, there is some $\hat{v}$ in $\mathcal{E}_B(b, pts, E)$ such that $\alpha(v) = \hat{v}$. Hence, there is some (p', b') in $\mathbf{post}_B(b, pts, p', S)$ such that $b' = b[\texttt{x} \mapsto \hat{v}]$. By definition, m' is in $\gamma_B(b')$.

- $S = \texttt{*x := } E$. If $m(\texttt{x})$ is undefined, not a location, or out of bounds, then $b(\texttt{x})$ is undefined, not a location, or out of bounds, and the claim is trivial. Assume $m(\texttt{x})$ is an in-bounds location l. Then, $m' = m[l \mapsto v]$ for some value v in $\mathcal{E}(m, E)$. Since m is in $\gamma_B(b)$ and $\gamma_{Pts}(pts)$, we have $b(\texttt{x}) = \alpha(l)$ and $pts(\texttt{x}, l)$. By Lemma 2.6, there is some $\hat{v}$ in $\mathcal{E}_B(b, pts, E)$ such that $\alpha(v) = \hat{v}$. Hence, there is some (p', b') in $\mathbf{post}_B(b, pts, p', S)$ such that $b' = b[l \mapsto \hat{v}]$. By definition, m' is in $\gamma_B(b')$.

- $S = \texttt{[}E\texttt{]}$. If $\mathcal{E}(m, E) = \bot$, then $\mathcal{E}_B(b, pts, E) = \bot$, by Lemma 2.7, and the claim is trivial. Thus, the only interesting case is when $\mathcal{E}(m, E) \neq \{0\}$ and $c = c'$. From Lemma 2.6, it follows that $\mathcal{E}_B(b, pts, E) \neq \{0\}$ and (p', b') is equal to (p, b). Hence, m' is in $\gamma_B(b')$.

This shows $\gamma_{Pts}(Q)$-soundness for $\widetilde{\mathcal{F}}_{\mathcal{B}}\langle Q\rangle$. $\square$

Lemma 2.10. *The analysis generator $\widetilde{\mathcal{B}}$ is sound.*

Proof. By Definition 7, Theorem 1.4, Lemma 2.8, and Lemma 2.9. □

Corollary 2.11. *If a points-to analysis $\mathcal{Q}$ is* SafeDeref*-sound, the composed memory safety analysis $\widetilde{\mathcal{B}} \circ \mathcal{Q}$ is* SafeDeref*-sound.*

Proof. By Lemma 2.10 and Theorem 1.6. □

Combining Corollary 2.11 with Theorems 2.2 and 2.3, we can compose $\widetilde{\mathcal{B}}$ with any of the points-to analyses described in Section 2.2 and the resulting analysis will be SafeDeref-sound. Recall from Theorem 1.3 that SafeDeref-soundness guarantees the detection of error states. If any non-SafeDeref state exists in $[\![\mathcal{P}]\!]_{\mathcal{C}}$, then a non-SafeDeref state is represented by the composed semantics; if only SafeDeref states are reachable in the composed analysis then no concrete non-SafeDeref state is reachable—the absence of error states can be proved.

2.5 Related Work

Methods for combining analyses have been described in the abstract interpretation community, starting with Cousot and Cousot [15]. The focus has been on exploiting mutual refinement to achieve the most precise combined analyses, as in Gulwani and Tiwari [30] and Cousot et al. [17]. The power domain of Cousot and Cousot [15, §10.2] provides a general model for analyses with conditional semantics. We believe our notion of conditional soundness provides a simpler model which captures the behavior of a variety of interesting analyses.

Pointer analysis for C programs has been an active area of research for decades [32, 24, 66, 3, 63, 27, 18, 31, 43]. The correctness arguments for points-to algorithms are typically stated informally—each of the analyses has been developed for the purpose of program transformation and understanding, not for use in a sound verification tool. Although Hind [32] proposes the use of pointer analysis in verification, the authors are not aware of any prior work that formally addresses the soundness of verification using points-to information.

Adams et al. [2] explored the use of Das' algorithm to prune the search space for a typestate checker and to generate initial predicates for a software model checker. In both cases, the use of the points-to information is essentially heuristic—the correctness of the overall approach does not depend on the points-to analysis being sound.

Dor, Rodeh, and Sagiv [22] describe a variation on traditional points-to analyses intended to improve precision for a sound, inter-procedural memory safety verifier. A proof of soundness is given in Dor's thesis [21]. However, the proof is not explicit about the obligations of the points-to analysis. We provide a more general framework for reasoning about verification using conditionally sound information.

Bruns and Chandra [11] provide a formal model for reasoning about pointer analysis based on transition systems. The focus of their work is primarily complexity and precision, rather than soundness.

Dhurjati, Kowshik, and Adve [20] define a program transformation which preserves the soundness of a flow-insensitive, equality-based points-to analysis (e.g., those of Steensgaard [63] and Lattner [43]) even for programs with memory safety errors. The use of an equality-based analysis is necessary to achieve an efficient implementation, but it limits the use of the technique in applications where a more precise analysis may be necessary, e.g., in verification. The soundness results we describe here are equally applicable to flow-sensitive, flow-insensitive, equality-based and subset-based pointer analyses.

Our abstraction for memory safety analysis is very similar to the formal models used in CCured [53] and CSSV [22]. Miné [50] describes a combined analysis for embedded control systems which incorporates points-to information. His analysis makes implementation-specific (i.e., unsound in general) assumptions about the layout of memory.

Chapter 3

The Cascade Verification Framework

Testing the ideas developed in this thesis required access to a flexible, powerful tool for software verification. In collaboration with other members of the Analysis of Computer Systems group at NYU (particularly Morgan Deters and Dejan Jovanović), I led the development of CASCADE, an open source, multi-language, multi-paradigm verification platform. CASCADE is suitable for a broad class of languages, ranging from low-level implementation languages, such as C, to high-level modeling languages, such as SPL [47, 48].

The current version of CASCADE is an total rewrite from a previous version developed by Nikhil Sethi and Clark Barrett [62]. It is implemented in Java using the CVC3 SMT solver [7] as the default back-end solver.

CASCADE is available for download from:

`http://cs.nyu.edu/acsys/cascade`

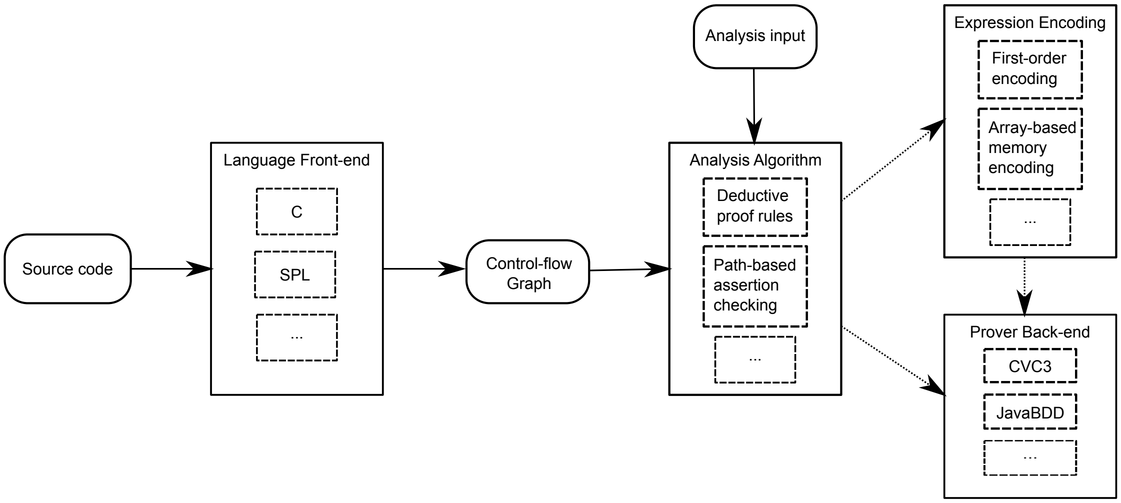


Figure 3.1: The design of CASCADE

3.1 Design Overview

Figure 3.1 illustrates the basic design of CASCADE. Source code is processed by a language front-end—implemented using the Rats! parsing framework [29]—into a generic control-flow graph (CFG) representation. The analysis algorithms take a domain-specific input (e.g., a deductive proof script, or a static path specification) and operate on the CFG. The analysis can make use of a variety of back-end provers and expression encodings.

The combination of different algorithms and encodings allows CASCADE to be used in a variety of different ways.

3.2 Cascade/C

CASCADE/C is a tool for precise static error analysis of C programs, intended for use in a *multi-stage analysis.* Since detailed, high-precision analysis scales relatively poorly with the size of the input program, we assume that a coarse,

```
void swap(int*x, int*y) {
  *x = *x + *y;
  *y = *x - *y;
  *x = *x - *y;
}
```

(a)

```
<controlFile>
  <sourceFile name="swap.c" id="1" />
  <run>
    <startPosition fileId="1" line="1" />
    <endPosition fileId="1" line="5">
      <assert><![CDATA[
         orig(*x)==*y && orig(*y)==*x
       ]]></assert>
    </endPosition>
  </run>
</controlFile>
```

(b)

Figure 3.2: Example using Cascade/C.

over-approximate analysis will be used to rule out most simple errors, relying on Cascade/C for errors where more precision is required.

Figure 3.2 illustrates the use of Cascade/C on a simple example. Figure 3.2(a) show the contents of the file `swap.c` and Fig. 3.2(b) is a control file describing a *run* to check in the code. The control file uses a simple XML syntax [67]. The run starts on Line 1 of the file (as specified by the `startPosition` tag) and ends on Line 7 (as specified by the `endPosition` tag). At the end of the run, Cascade/C will check the condition contained in the `assert` tag: that the final value of `*y` is equal to the initial value of `*x` (i.e., the value at the start of the run) and that the final value of `*x` is equal to the initial value of `*y`. The body of the assertion is embedded in a `CDATA` section so that it can use standard C syntax without XML escapes.

Cascade supports several different encodings for expressions and paths. For example, arithmetic expressions can be encoded using either unbounded integers or fixed-size bit vectors; the semantics of paths can be encoded using first-order formulas to represent the strongest post-condition or using functional expressions to represent a state transformer. The encodings can be combined according to the user's preference.

$$\begin{aligned}
m_1 = m_0[m_0[\texttt{\&x}] \mapsto m_0[\texttt{\&x}] + m_0[\texttt{\&y}]] \wedge \\
m_2 = m_1[m_1[\texttt{\&y}] \mapsto m_1[\texttt{\&x}] - m_1[\texttt{\&y}]] \wedge \\
m_3 = m_2[m_2[\texttt{\&x}] \mapsto m_2[\texttt{\&x}] - m_2[\texttt{\&y}]] \implies \\
m_0[m_0[\texttt{\&x}]] = m_3[m_3[\texttt{\&y}]] \wedge m_0[m_0[\texttt{\&y}]] = m_3[m_3[\texttt{\&x}]]
\end{aligned}$$

(a) First-order encoding

$$\begin{gathered}
(\lambda m. m_0[m_0[\texttt{\&x}]] = m[m[\texttt{\&y}]] \wedge m_0[m_0[\texttt{\&y}]] = m[m[\texttt{\&x}]]) \\
((\lambda m. m[m[\texttt{\&x}] \mapsto m[\texttt{\&x}] - m[\texttt{\&y}]]) \\
((\lambda m. m[m[\texttt{\&y}] \mapsto m[\texttt{\&x}] - m[\texttt{\&y}]]) \\
((\lambda m. m[m[\texttt{\&x}] \mapsto m[\texttt{\&x}] + m[\texttt{\&y}]])\ m_0)))
\end{gathered}$$

(b) Functional encoding

Figure 3.3: CASCADE encodings of the path in Fig. 3.2

Figure 3.3 illustrates two encodings for the path in Fig. 3.2. We use `&x` and `&y` to denote the location of variables `x` and `y`, respectively, in memory. The encoding of Fig. 3.3(a) represents the path using a first-order formula. The assertion is valid if it is implied by the strongest post-condition of the path. The changing state is represented using fresh variables (m_0, m_1, etc.). The encoding of Fig. 3.3(b) represents the path as a function encoding the state transformation; the assertion is valid if it satisfied by any state produced by the transformer. In both cases, we omit background assumptions necessary to avoid spurious counterexamples (e.g., that `&x` and `&y` are distinct).

Note that the encodings in Fig. 3.3 are essentially equivalent. However the back-end prover may treat equalities differently from functional transformations; this may affect performance. The encoding will also affect the form of the counterexample produced for invalid assertions. For example, in our experience, the functional encoding yields better performance and more compact counterexamples using the Cvc3 back-end.

3.3 Cascade/Spl

Cascade/Spl is a tool for deductive verification of safety and liveness properties of programs in the Simple Programming Language (Spl). The use of deductive verification allows the tool the flexibility to support infinite-state and parameterized Spl programs; it also places some burden on the user to properly direct the tool using invariants and ranking functions. Cascade/Spl was originally intended to support a graduate course in deductive verification at NYU, replacing the use of Tlv in earlier offerings of the course. The tool is unique because it combines the following three features:

- **Automated verification.** Cascade/Spl supports fully automatic proof generation using state-of-the-art SMT solver back-ends. This verification is performed on high-level Spl programs, rather than low-level models.
- **Support for parameterized systems.** Cascade/Spl supports parameterized systems, in which the number of parallel execution processes is not known *a priori* (e.g., a token ring system with N processes, where N is unbounded).
- **Open-source and extensible.** Cascade/Spl is implemented as part of the Cascade platform. Cascade/Spl can serve as an example for the development of other useful language modules. In our experience, students have been able to make useful contributions to the system over the course of a single semester.

$$y : \ \textbf{integer where } y = 1$$

$$\|_{i=1}^{N} P[i] :: \left[\begin{array}{l} \ell_0 : \ \textbf{loop forever do} \\ \qquad \left[\begin{array}{ll} \ell_1 : & \textbf{Non-critical} \\ \ell_2 : & \textbf{request } y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release } y \end{array} \right] \end{array} \right]$$

(a)

$$\begin{array}{rcll} \Omega & \Longrightarrow & \varphi & \text{(I1)} \\ \varphi \wedge \rho & \Longrightarrow & \varphi' & \text{(I2)} \\ \varphi & \Longrightarrow & p & \text{(I3)} \\ \hline \multicolumn{4}{c}{\Box\, p} \end{array}$$

(b)

Figure 3.4: The MUX-SEM program for N processes.

CASCADE/SPL currently supports a subset of the SPL language, including basic types, arrays, parameterized processes and sub-processes. Implementation of additional features (e.g., modules, lists, and channels) is ongoing work.

3.3.1 SPL

Figure 3.4(a) illustrates a simple parameterized SPL program: MUX-SEM over N processes. The mutual exclusion property for this program is

$$\forall i, j : \quad i \neq j \ \wedge \ at\_\ell_3[i] \implies \neg at\_\ell_3[j] \tag{3.1}$$

where $at\_\ell_3[i]$ is the predicate that holds when the program counter of process i is at ℓ_3. It is well known (e.g., [48, §1.2]) that (3.1) is not an inductive invariant of MUX-SEM; however, the invariance of (3.1) can be established by an *inductive strengthening*, i.e., an inductive invariant φ that implies (3.1), using the deductive rule INV (Fig. 3.4(b)). The rule states, simply, that if: (I1) φ holds in the initial states, (I2) is preserved by the transition relation, and (I3) implies p, then p is invariant ($\Box\, p$).

One possible inductive strengthening of (3.1) is:

$$y \geq 0 \quad \wedge \quad [\, \forall i, j :\ i \neq j \implies at\_\ell_{3,4}[i] + at\_\ell_{3,4}[j] + y = 1 \,] \tag{3.2}$$

This invariant has the advantage of belonging to the *array property fragment* of the theory of arrays [10], for which there is a complete decision procedure implemented in CVC3. (In general, invariants expressed in a complete fragment will guarantee correct counterexamples, which in turn is crucial in guiding the user toward correct invariants.)

In order to check the invariant (3.1) using INV, CASCADE/SPL first builds the *fair discrete system* (FDS) [47, 48] for MUX-SEM, which collects the state variables of MUX-SEM, its initial state Ω, its transition relation ρ, and the *justice* (weak fairness) and *compassion* (strong fairness) requirements. The components of the FDS are encoded as first-order formulas. We use the components to construct queries validating the premises of the INV rule applied to the given invariant and inductive strengthening (Fig. 3.5). CASCADE/SPL can prove mutual exclusion for MUX-SEM using the rule INV with the strengthening (3.2) in less than 5 seconds.

3.3.2 Related work

CASCADE/SPL is intended as a successor to TLV [58]. TLV allows for both deductive and algorithmic verification of finite systems expressed as SMV models [49]. CASCADE/SPL focuses on deductive verification, and can handle infinite-state and parameterized systems expressed as high-level SPL programs. CASCADE/SPL does not yet have a high-level scripting language like TLV-BASIC; instead, verification goals are expressed through a Java API.

```
public static ValidityResult inv(StateProperty p,
                                 StateProperty phi,
                                 TransitionSystem tsn) {
    StateProperty i1 = tsn.initialStates().implies(phi);
    ValidityResult res = tsn.checkValidity(i1);

    if (!res.isValid()) {
        System.out.println("Premise I1 is not valid.");
        return res;
    }

    StateProperty i2 = phi.and(tsn.transitionRelation()).implies(phi.prime());
    res = tsn.checkValidity(i2);
    if (!res.isValid()) {
        System.out.println("Premise I2 is not valid.");
        return res;
    }

    StateProperty i3 = phi.implies(p);
    res = tsn.checkValidity(i3);
    if (!res.isValid()) {
        System.out.println("Premise I3 is not valid.");
        return res;
    }

    System.out.println("* * * Assertion p is invariant.\n");
    return res;
}
```

Figure 3.5: A portion of the implementation of Inv in Cascade/Spl.

The STeP [8] and Pvs [56] tools provide deductive verification facilities in an interactive setting. In contrast, Cascade/Spl is designed to provide fully automated operation. In practice, this means that the proofs generated by Cascade/Spl are limited by the completeness of the prover's decision procedures.

Other tools for deductive verification include Krakatoa [25], for Java programs, and Caduceus [25] and Jessie [52], for C programs. Cascade/Spl handles higher-level specifications in Spl, which allows for reasoning about parallelism and parameterized systems.

Chapter 4

Verifying Low-Level Datatypes

Packet-level networking code is critical to communications infrastructure and vulnerable to malicious attacks. This code is typically written in low-level languages like C or C++. Packet fields are "parsed" using pointer arithmetic and bit-wise operators to select individual bytes and sequences of bits within a larger untyped buffer (e.g., a `char` array). This approach yields high-performance, portable code, but can lead to subtle errors.

An alternative is to write packet-processing code in special-purpose high-level languages, e.g., binpac [57], Melange [46], Morpheus [1], or Prolac [38]. These languages typically provide a facility for describing network packets as a set of nested, and possibly recursive, datatypes. The language compilers then produce low-level packet-processing code which aims to match or exceed the performance of the equivalent hand-coded C/C++. This requires an expensive commitment to rewriting existing code.

We propose a new approach, one which fuses the power of higher-level datatypes with the convenience and efficiency of legacy code. The key idea is to use a high-level description of "packet types" as the basis for a *specification*, not an *imple-*

mentation. Instead of using a compiler to try to reproduce a performant implementation, we can annotate the existing implementation to indicate the intended high-level semantics, then verify that the implementation is consistent with those semantics. We make use of the theories of inductive datatypes, bit vectors, and arrays in Cvc3 to encode the relationship between the high-level and low-level semantics. Using this encoding, it is possible to verify that the low-level code represents, in essence, an implementation of a well-typed high-level specification.

In this chapter, we will present our proposed notation for defining packet datatypes and stating datatype invariants in C code. We describe the translation of the datatype definition and code assertions into verification conditions in the Cvc3 SMT solver. The encoding relies crucially on automatically generated separation invariants, which allow Cvc3 to efficiently reason about recursive data structures without producing false assertion failures due to spurious aliasing relationships. Finally, we present a case study applying our approach to real code from the BIND DNS server. We are able to verify high-level data invariants of the code with reasonable efficiency. To our knowledge, no other verification tool is capable of automatically proving such datatype invariants on existing C code.

4.1 A Motivating Example

Figure 4.1(a) illustrates the definition of a simple, high-level list datatype in a notation similar to that of languages like ML and Haskell. The type has two constructors: `cons`, which creates a list node with an associated `data` array and a `cdr` field representing the remainder of the list, and `nil`, which represents an empty list. Figure 4.1(b) gives the high-level pseudo-code for a function that computes the length of a list, defined as the number of `cons` values encountered via `cdr`

```
type List =
  cons {
    count: Nat,
    data: Int array,
    cdr: List
  }
| nil
```

(a)

```
Nat list_length(List lst) {
    Nat count = 0;
    while( isCons(lst) ) {
        count++;
        lst = cdr(lst);
    }
    return count;
}
```

(b)

```
type List =
  cons {
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }
| nil {
    tag:8 = 0x00
  }
```

(c)

```
u_int list_length(const u_char *p) {
    u_int n, count = 0;
    while( (n = *p++) & 0x80 ) {
        { isCons(prev(p)) }
        count++;
        p += n & 0x7f;
        { toList(p) = cdr(prev(p)) }
    }
    if( n != 0 ) // malformed list
        return (-1);
    { isNil(p) }
    return count;
}
```

(d)

Figure 4.1: Defining and using a simple linked list datatype.

"links" before a `nil`. The code simply checks whether `lst` is a `cons` value using the "tester" function `isCons`. If it is, it increments the length and updates `lst` using the `cdr` field. If it is not, it returns the computed length.

In a high-level language, the compiler is given the freedom to implement datatypes like `List` as it chooses, typically using linked heap structures to represent individual datatype values. The programmer concentrates on the high-level semantics of the algorithm, allowing the compiler to encode and decode the data. By contrast, in packet processing code, the datatype is defined in terms of an explicit data layout. The data is "packed" into a contiguously allocated block of memory.

The high-level algorithm and the encoding and decoding of data are intertwined.

The `List` type in Fig. 4.1(c) illustrates a simple “packed” linked list implementation. Like the definition in Fig. 4.1(a), `List` is a union type with two variants. However, instead of simply declaring a set of data fields, each variant explicitly defines its own representation. The representation of a `cons` value is: a 1-bit `tag` field (the highest-order bit of the first byte), a 7-bit `count` field (the lower-order bits of the first byte), a `data` field of exactly `count` bytes, and another `List` value `cdr`, which follows immediately in memory. The value of `tag` is constrained by the constant bit vector value `0b1`. The constraint requires the `tag` bit of a `cons` value always to be `1`. The representation of a `nil` value has a similar constraint: a `nil` value consists of a single 8-bit `tag` field, which must be `0x00`. The fact that the `tag` bit of a `cons` value must be `1` while the bits of a `nil` value must all be `0` ensures that we can unambiguously decode `cons` and `nil` values. (A full grammar for “packed” datatype definitions is given in Section 4.2.1.)

Figure 4.2 illustrates the interpretation of a sequence of bytes as a `List` value. The first byte (`0x82`) has its high bit set; thus, it is a `cons` value. The low-order bits tell us that `count` is `2`; thus, `data` has two elements: `0x01` and `0x02`. The `cdr` field is another `List` value, encoded starting at the next byte. This byte (`0x81`) is also a `cons` value, since it also has its high bit set. Its `count` field is `1`, its `data` field the single element `0x03`. Its `cdr` is the `List` value at the next byte (`0x00`), a `nil` value.

Figure 4.1(d) gives a low-level implementation of the length function, which operates over the implicit `List` value pointed to by the input `p`. (The bracketed, italicized portions of the code are verification annotations, which are described in Section 4.2.3.) Note that the structure of the function is very similar to the

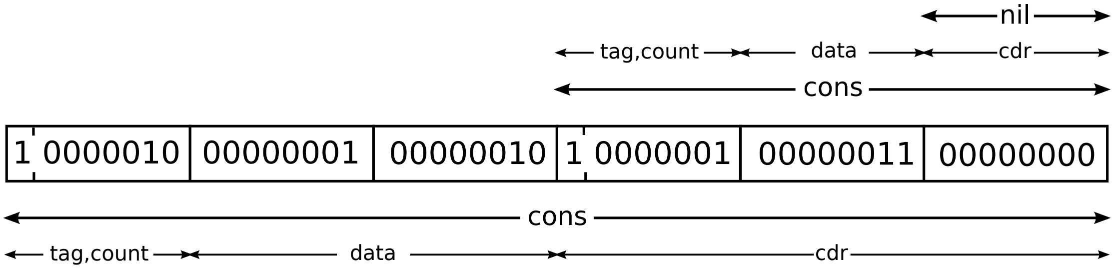


Figure 4.2: The layout of a `List` value.

code in Fig. 4.1(b), but that high-level operations have been replaced by their low-level equivalents—pointer arithmetic and bit-masking operations are used to detect constructors and select fields. A notable addition is the `if` statement that appears after the `while` loop. In the high-level code, we could assume that the data was well-formed, i.e., that every list is either a `cons` or a `nil` value. In the low-level implementation, we may encounter byte sequences which are not assigned a meaning by the datatype definition—in this case a non-zero byte in which the high bit is not set, which satisfies the data constraints of neither `cons` nor `nil`. The function handles this erroneous case by returning an error code.

The challenge, in essence, is to prove that the low-level code in Fig. 4.1(d) is a refinement of the high-level code in Fig. 4.1(b). To this end, we need to build a bridge between the high-level semantics of the datatype and the low-level implementation.

4.2 Our Approach

The verification process proceeds in four steps:

1. The programmer provides a datatype declaration, as in Fig. 4.1(c), defining the high-level structure and layout of the data.

2. Using the datatype declaration, we generate a set of Cvc3 declarations and axioms encoding the relationship between the high-level type and its implementation.

3. The programmer adds code annotations specifying the expected behavior of the low-level code, in terms of functions derived from the datatype definition.

4. We use the Cascade verification platform to translate the code and annotations into a set of verification conditions to be checked by Cvc3. If all of the verification conditions are valid, then the code satisfies the specification.

4.2.1 Datatype definition

Figure 4.3 gives the full grammar for datatype definitions. The notation for datatype definitions is similar to that of disjoint union types in higher-level languages like ML and Haskell. There is an important distinction: unlike datatype implementations generated by compilers, it is up to the user to ensure that the encoding of values is unambiguous and consistent. The declaration should provide all of the information needed both to encode a datatype value as a sequence of bytes and to decode a well-formed sequence of bytes as a high-level datatype value.

A type consists of a set of constructors. Each constructor has a set of fields. A field type is one of four kinds: a bit vector of constant integer size, a plain C scalar type, an array of C type elements, or another datatype. (The syntax of C type declarators is that of ANSI/ISO C [4].) Bit vectors and C types may have value constraints. Bit vector constants are preceded by `0b` (for binary constants) or `0x` (for hexadecimal constants). Arrays have a length: either a constant integer or the value of a prior field—the declaration language supports a limited form of

$$\begin{aligned}
\mathit{Type} &::= \texttt{type}\ \mathit{Id} \,\texttt{=}\, \mathit{Cons}\ (\texttt{|}\ \ \mathit{Cons})* \\
\mathit{Cons} &::= \mathit{Id}\, \{\ \ \mathit{Field}\ (\texttt{,}\ \mathit{Field})* \} \\
\mathit{Field} &::= \mathit{Id} : \mathit{FieldType} \\
\mathit{FieldType} &::= \mathit{BvType} \mid \mathit{CType} \mid \mathit{ArrType} \mid \mathit{TypeId} \\
\mathit{BvType} &::= \mathit{IntConst}\ (\texttt{=}\, \mathit{BvConst})? \\
\mathit{BvConst} &::= \texttt{0b}[\texttt{01}]+ \mid \texttt{0x}[\texttt{0-9a-fA-F}]+ \\
\mathit{CType} &::= \mathit{CScalarType}(\texttt{=}\ \mathit{CConst})? \\
\mathit{ArrType} &::= \mathit{CType}\ \texttt{[}\mathit{ArrLength}\texttt{]} \\
\mathit{ArrLength} &::= \mathit{IntConst} \mid \mathit{Id} \\
\mathit{TypeId} &::= \mathit{Id}
\end{aligned}$$

Figure 4.3: Grammar for datatype definitions.

dependent types.

4.2.2 Translation to Cvc3

It is straightforward to translate the datatype definition into an inductive datatype in the input language of Cvc3. The translation for the `List` datatype is given in Fig. 4.4. We use $\mathbb{Z}^+$ to denote the type of natural numbers; $\mathcal{BV}_k$ to denote the type of bit vectors of size k (i.e., k-tuples of booleans); and (α, β) array to denote the type of arrays with indices of type α and elements of type β. We use N to denote the (platform-dependent) size of a pointer (i.e., the type of pointers is $\mathcal{BV}_N$). For an array a, $a[i]$ denotes the element of a at index i; similarly, for a bit vector b, $b[i]$ denotes the ith bit of b and $b[j{:}i]$ denotes the *extraction* of bits i through j (the result is a bit vector of size $j - i + 1$). The size of the result of arithmetic operations on bit vectors is the size of the larger operand; the smaller operand is implicitly zero-extended. When used in an integer context, bit vectors are interpreted as unsigned.

The translation produces a Cvc3 datatype definition reflecting the data layout

$$\begin{aligned}\textbf{datatype}\ \ \mathit{List} = \ & \mathit{cons}\ \{\ \mathit{count} : \mathcal{BV}_7,\ \mathit{data} : (\mathcal{BV}_N, \mathcal{BV}_8)\ \textsf{array},\ \mathit{cdr} : \mathit{List}\ \} \\ \mid\ & \mathit{nil} \\ \mid\ & \mathit{undefined}\end{aligned}$$

$$\begin{aligned}\mathit{toList} &: (\mathcal{BV}_N, \mathcal{BV}_8)\ \textsf{array} \times \mathcal{BV}_N \to \mathit{List} \qquad & m &: (\mathcal{BV}_N, \mathcal{BV}_8)\ \textsf{array} \\ \mathit{sizeOfList} &: \mathit{List} \to \mathbb{Z}^+ & \ell &: \mathcal{BV}_N\end{aligned}$$

$$\begin{aligned}&\textbf{let}\ x = \mathit{toList}(m, \ell)\ \textbf{in} \\ &\quad \mathit{isCons}(x) \iff m[\ell][7] = 1 && (\textsc{ConsTest}) \\ &\quad \mathit{isNil}(x) \iff m[\ell] = 0 && (\textsc{NilTest}) \\ &\quad \mathit{isCons}(x) \implies \mathit{count}(x) = m[\ell][6{:}0] \\ &\qquad\qquad \wedge\ (\forall 0 \le i < \mathit{count}(x).\ \ \mathit{data}(x)[i] = m[\ell + i + 1]) \\ &\qquad\qquad \wedge\ \mathit{cdr}(x) = \mathit{toList}(m, \ell + \mathit{count}(x) + 1) \\ &&& (\textsc{ConsSel})\end{aligned}$$

$$\begin{aligned}\mathit{sizeOfList}(\mathit{cons}(\mathit{count}, \mathit{data}, \mathit{cdr})) &= 1 + \mathit{count} + \mathit{sizeOfList}(\mathit{cdr}) && (\textsc{ConsSize}) \\ \mathit{sizeOfList}(\mathit{nil}) &= 1 && (\textsc{NilSize}) \\ \mathit{sizeOfList}(\mathit{undefined}) &= 0 && (\textsc{UndefSize})\end{aligned}$$

Figure 4.4: Datatype definition and axioms for the type `List`

of the declaration augmented with an explicit *undefined* value. Note that the `tag` fields are omitted from the definition—since they are constrained by constants, they are only needed to decode the high-level data value.

Cvc3 automatically generates a set of datatype testers and field selectors. The testers *isCons*, *isNil*, and *isUndefined* are predicates that hold for a *List* value x iff x is, respectively, a *cons*, *nil*, or *undefined* value. The selectors *count*, *data*, and *cdr* are functions that map a *List* value to the value of the corresponding fields.

Note that the definition of *List* itself does not include any data constraints on field values. These constraints are introduced by the function *toList*, which maps a pointer-indexed array of bytes m and a location ℓ to the *List* value represented by the sequence of bytes starting at ℓ in m. The axioms ConsTest and NilTest enforce the data constraints on the `tag` fields of `cons` and `nil`, respectively. The axiom ConsSel represents the encoding of the remaining fields of `cons`. Note that there is no explicit rule for the value *undefined*: if the data constraints given in ConsTest and NilTest do not apply, then the only remaining value that *toList* can return is *undefined*.

The function *sizeOfList* maps a *List* value to the size of its encoding in bytes. By convention, the size of *undefined* is 0.

4.2.3 Code assertions

The functions generated by the Cvc3 translation are exposed in the assertion language as functions that take a single pointer argument. In the case of the function `toList`, the additional array argument, representing the configuration of memory, is introduced in the verification condition translation. The pointer argument of the other functions is implicitly converted to a `List` value using `toList`. The assertion language also provides auxiliary functions `init` and `prev`, mapping variables to their initial values in, respectively, the current function and loop iteration.

Returning to the code in Fig. 4.1(d), the bracketed, italicized assertions state the expected high-level semantics of the implementation. Specifically, they assert:

- The loop test succeeds only for `cons` values.
- The body of the loop sets `p` to the `cdr` of its initial value in each loop iteration.
- If the value is well-formed, then `p` points to a `nil` value when the function returns.

The functions representing testers rely on the data constraints of the type, e.g., `p` points to a `cons` value iff the byte sequence pointed to by `p` satisfies the data constraints of `cons` (i.e., the high bit of `*p` is set). The functions representing testers rely on the structure of the type, e.g., `toList(q)==cdr(p)` iff `p` points to a `cons` value and `q==p+count(p)+1`.

Loops can be annotated with invariants: we can separately prove initialization and preservation of the invariant, and that each assertion in the body of the loop is valid when the invariant is assumed on entry.

4.2.4 Verification condition generation

The final verification step is to use the CASCADE verification platform to translate the code and assertions into formulas that can be validated by CVC3. Verification is driven by a *control file*, which defines a set of paths to check and allows annotations and assertions to be injected at arbitrary points along a path. Each code assertion is transformed into a verification condition, which is passed to CVC3 and checked for validity. For each condition, CVC3 will return "valid" (the condition is always true), "invalid" (the condition is not always true), or "unknown" (due to incompleteness, CVC3 could not prove invalidity). CASCADE returns "valid" for a path iff CVC3 returns "valid" for every assertion on the path. If CVC3 returns "invalid" or "unknown" for any assertion, CASCADE returns "invalid", along with a counterexample.

Note 4.1. Since the background axioms that define datatypes are universally quantified, deciding validity of the generated verification conditions is undecidable in general. CVC3 will never return "invalid" for any verification condition that it cannot prove valid; instead, it will return "unknown" when a pre-determined instantiation limit is reached. There are fragments of first-order logic that are decidable with instantiation-based algorithms [28]. Encoding the datatype assertions in a decidable fragment of first-order logic is a subject for future work. □

Cascade supports a number of encodings for C expressions and program semantics. For datatype verification, we make use of a bit vector encoding, which is parameterized by the platform-specific size of a pointer and of a memory word.

An additional consideration is the memory model used in the verification condition. The memory model specifies the interpretation of pointer values and the effect of memory accesses (both reads and writes) on the program state. A memory model may abstract away details of the program's concrete semantics (e.g., by discarding information about the precise layout of structures in memory) or it may refine the concrete semantics (e.g., by choosing a deterministic allocation strategy). We discuss the memory model in detail in the next section.

4.3 Memory Model

In order to accurately reflect the datatype representation, we require a memory model that is bit-precise. At the same time, to avoid a blow-up in verification complexity and overly conservative results, we would like a relatively high-level model that preserves the separation invariants of the implementation. To this end, we define a memory model based on separation analysis [33, 59] that we call a *partitioned heap.*

The flat model. First, we will define for comparison a simple model which is self-evidently sound. A *flat memory model* interprets every pointer expression as a bit vector of size N. Every allocated object in the program is associated with a region of memory (i.e., a contiguous block of locations) distinct from all previously allocated regions. The state of memory is modeled by a single pointer-indexed array m. The value stored at location ℓ is thus $m[\ell]$.

Using the flat memory model, we can translate the first assertion in Fig. 4.1(d) into the verification condition

$$\begin{aligned} &m_1 = m_0[\texttt{\&p} \mapsto m_0[\texttt{\&p}] + 1] \wedge \\ &\qquad\qquad m_2 = m_1[\texttt{\&n} \mapsto m_0[m_0[\texttt{\&p}]]] \wedge m_2[\texttt{\&n}][7] \implies \\ &\qquad\qquad\qquad\qquad \mathit{isCons}(\mathit{toList}(m_2, m_0[\texttt{\&p}])) \end{aligned}$$

where we use `&x` to denote the location in memory of the variable `x` (i.e., its *lvalue*) and $a[i \mapsto e]$ to denote the update of array a with element e at index i. Assuming `&p`, `&n`, and $m[\texttt{\&p}]$ are distinct, the validity of the formula is a direct consequence of the axiom ConsTest.

The flat model accurately represents unsafe operations like casts between incompatible types and bit-level operations on pointers. However, it is a very weak model—its lack of guaranteed separation between objects makes it difficult to prove strong properties of data-manipulating programs.

Example 4.1. Consider the Hoare triple

$$\{\ \texttt{toList(q)==cdr(p)}\ \}\ \texttt{i++}\ \{\ \texttt{toList(q)==cdr(p)}\ \}$$

where `p` and `q` are known to not alias `i`. In a flat memory model, this is interpreted as

$$\mathit{toList}(m_0, m_0[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(m_0, m_0[\texttt{\&p}])) \wedge \\ m_1 = m_0[\texttt{\&i} \mapsto m_0[\texttt{\&i}] + 1] \implies \\ \mathit{toList}(m_1, m_1[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(m_1, m_1[\texttt{\&p}]))$$

Since *toList* is defined axiomatically using recursion (see Fig. 4.4), it is not immediately obvious that the necessary lemma

$$\mathit{toList}(m_0, m_0[\texttt{\&p}]) = \mathit{toList}(m_1, m_1[\texttt{\&p}])$$

is implied (similarly for `q`). Even if `p` and `q` can never point to `i`, we cannot rule out the possibility that the `List` values pointed to by `p` and `q` depend in some way on the value of `i`. Now, suppose we add the assumption

`allocated(p,p+sizeOfList(p))`,

where `allocated(x,y)` means that pointer `x` is the base of a region of memory, disjoint from all other allocated regions, bounded by pointer `y`. Even then, the proof of the assertion relies on the following theorem, which is beyond the capability of automated theorem provers like Cvc3 to prove:

$$(\forall y : x \leq y \leq x + \mathit{sizeOfList}(\mathit{toList}(m_0, x)) : m_0[y] = m_1[y]) \implies \\ \mathit{toList}(m_0, x) = \mathit{toList}(m_1, x)$$

What we require is a separation invariant allowing us to apply the "frame rule" of separation logic [60, 54]:

$$\{\ \texttt{toList(q)==cdr(p)*i==}v\ \}\ \texttt{i++}\ \{\ \texttt{toList(q)==cdr(p)*i==}v+1\ \}$$

where `*` denotes *separating conjunction*: $A{*}B$ holds iff memory can be partitioned into two disjoint regions R and R' where A and B hold, respectively.

The partitioned model. The separation invariants we need can be obtained using separation analysis [33, 59]. The analysis can be understood as the inverse of *may-alias analysis* [40, 41]: if pointers `p` and `q` can never alias, then the objects they point to must be separated (i.e., they occupy disjoint regions of memory).

The output of the separation analysis is a *partition* $P = \{P_1, \ldots, P_k\}$, where each P_i represents a disjoint region of memory, and a map from pointer expressions to regions—if expression E is mapped to partition P_i, then E can only point to objects allocated in region P_i. If the separation analysis maps pointers expressions E and E' to different partitions, then E and E' cannot be aliased in any well-defined execution of the program.

A *P-partitioned memory model* for partition $P = \{P_1, \ldots, P_k\}$ interprets every pointer expression as a pair $(\ell, i) \in \mathcal{BV}_N \times \mathbb{Z}^+$, where ℓ is a location and i is a partition index. The state of memory is modeled by a collection of pointer-indexed arrays $\langle m_1, \ldots, m_k \rangle$. The location pointed to by pointer expression (ℓ, i) is the array element $m_i[\ell]$.

Example 4.2. The program in Fig. 4.1(d) can be divided into two partitions. The first partition contains the parameter `p` and local variables `n` and `size`. The second partition contains the object pointed-to by `p`. We represent the two partitions by two memory arrays, s and h, respectively. Thus, the value of the variable `n` is represented by the array element $s[\texttt{\&n}]$; the value of the expression `*p` is represented by the array element $h[s[\texttt{\&p}]]$.

A partitioned memory model solves the problem of Example 4.1 by isolating the *List* value in its own partition:

$$
\begin{aligned}
h_0 \neq s_0 \wedge \mathit{toList}(h_0, s_0[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(h_0, s_0[\texttt{\&p}])) \wedge \\
s_1 = s_0[\texttt{\&i} \mapsto s_0[\texttt{\&i}] + 1] \implies \\
\mathit{toList}(h_0, s_1[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(h_0, s_1[\texttt{\&p}]))
\end{aligned}
$$

Given that `&p`, `&q` and `&i` are distinct, the formula is trivially valid.

We say a program is *memory safe* if all reads and writes through pointers occur only within allocated objects. Like pointer analysis, the soundness of the separation analysis is conditional on memory safety. Thus, the soundness of verification using a partitioned memory model will likewise be conditional on memory safety.

It may seem questionable to attempt to verify a program using information which depends for its correctness on prior verification of the same program. In the next section, we will show that a SAFEDEREF-sound combination is possible. It is thus essential that the verification conditions include assertions that establish the memory safety of the statements along each path in the program.

In our experience, a partitioned memory model can make an order-of-magnitude difference in verification time compared to a flat memory model—indeed, properties are provable by CVC3 using a partitioned model that cannot be proved using a flat model (see Section 4.5.1).

4.4 Soundness

We will now consider the soundness of the partitioned memory model. For the purposes of this section, we will set aside the bit-precise semantics and return to the more abstract semantics of Section 2.1.

We will define a *partitioned analysis* which takes as input the results of a separation analysis. The separation analysis will be used to split the memory state into a *collection* of distinct memories and to assign each pointer expression in the program a unique memory to which it refers. We can thus isolate the effects of memory operations and simplify the verification process.

Note 4.2. Demonstrating the soundness of the bit-precise interpretation is a simple matter of projecting the set Vals to bit vectors of size N in both the concrete and abstract semantics. Since integers and locations would not necessarily be structurally distinct, this would require stricter type safety assumptions. Since the projection would not necessarily be injective in the case of out-of-bounds locations, SafeDeref would be a trace property, ensuring that no location value is computed using ill-defined operations. We beg the reader's indulgence in eliding these complex but inconsequential details. □

4.4.1 Separation Analysis

A *separation environment* R is a triple $(\mathcal{R}, \mathbf{region}, \mathbf{rpoint})$, where:

- $\mathcal{R}$ is a finite set of *memory regions*;
- $\mathbf{region} : \mathbb{H} \to \mathcal{R}$ maps homes to regions; and
- $\mathbf{rpoint} : \mathcal{V} \to \mathcal{R}$ is a partial function mapping variables to the regions they point to.

We lift **region** to map locations in the natural way: $\mathbf{region}(h[i]) = \mathbf{region}(h)$. When it is convenient, we use $\mathbf{region}_R$ and $\mathbf{rpoint}_R$ to refer to the respective components of the separation environment R. We denote the set of separation environments by *Sep*.

The concretization function γ_{Sep} maps a separation relation to the set of memories where the separation invariants hold.

$$\gamma_{\mathit{Sep}}(R) = \{m \mid \forall \mathtt{x}, r, l : \mathbf{rpoint}_R(\mathtt{x}) = r \wedge m(\mathtt{x}) = l \ \wedge l \text{ is in-bounds } \implies \mathbf{region}_R(l) = r\}$$

The set of separation states is the set of pairs (p, R), where $p \in \mathcal{L}$ represents a program position and R is a separation environment. The result of a separation analysis is a set of separation states.

We say a set of separation states Q is *well-formed* if the following properties hold:

- The set of regions $\mathcal{R}$ is the same for every environment in Q.
- The map **region** is the same for every environment in Q.
- There is at most one separation state for each program position in Q.

Well-formedness guarantees that the set of states is consistend—for example, it prevents the analysis from "moving" a variable from one region to another at different program locations. In the remainder of this chapter, we will assume that all sets of separation states are well-formed.

4.4.2 The Partitioned Analysis

The partitioned analysis will take as input set of separation states Q and will use as its state, instead of a single memory, a *vector* of memories, one for each region r in the separation environments of Q. We denote by $\vec{m}^R_h$ the memory associated with the region $\mathbf{region}_R(h)$ in vector $\vec{m}$. We extend this notation to locations and variables in the natural way, e.g., $\vec{m}^R_l$ and $\vec{m}^R_{\mathtt{x}}$. We denote by $\vec{m}^R_{*\mathtt{x}}$ the memory associated with the region $\mathbf{rpoint}_R(\mathtt{x})$ in $\vec{m}$.

The concretization function γ_R maps a vector of memories $\vec{m}$ to the set of memories that agree with $\vec{m}^R_l$ at all in-bounds locations l. I.e.,

$$\gamma_R(\vec{m}) = \{m \mid \forall l \in \mathbb{L} : l \text{ is in-bounds } \implies m(l) = \vec{m}^R_l(l)\}$$

Figure 4.5 defines the interpretations $\mathcal{E}_K$ and $\mathbf{post}_K$ of, respectively, expressions and statements. The operators $\widetilde{\oplus}$ and $\widetilde{\trianglelefteq}$ denote the semantic operators, just as in Fig. 2.3 (page 20).

We now define the partitioned analysis, parameterized by a set of separation states.

Definition 12. The analysis generator $\widetilde{\mathcal{K}}$ maps a set of separation environments Q to the partitioned analysis $\widetilde{\mathcal{K}}\langle Q\rangle$ defined by the parameterized interpretations

$$\widetilde{\mathcal{I}}_{\mathcal{K}}\langle Q\rangle[\mathcal{P}] = \{(en, \vec{m}) \mid \forall l \in \mathbb{L}, m \in \vec{m} : m(l) \text{ is not a location}\}$$

$$\widetilde{\mathcal{F}}_{\mathcal{K}}\langle Q\rangle[\mathcal{P}](p, \vec{m}) = \bigcup_{(p,S,p')\in\mathcal{T}} \bigcup_{(p,R)\in Q} \mathbf{post}_K(\vec{m}, R, p', S)$$

The concretization function for the resulting analysis, $\gamma_{\widetilde{\mathcal{K}}\langle Q\rangle}(\vec{m})$, is defined as

$$\mathcal{E}_K(\vec{m}, R, n) = \{n\}$$

$$\mathcal{E}_K(\vec{m}, R, \mathtt{x}) = \begin{cases} \mathbb{Z}, & \text{if } \vec{m}^R_{\mathtt{x}}(\mathtt{x}) \text{ is undefined} \\ \{\vec{m}^R_{\mathtt{x}}(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_K(\vec{m}, R, {*}\mathtt{x}) = \begin{cases} \bot, & \text{if } \vec{m}^R_{\mathtt{x}}(\mathtt{x}) \text{ is undefined, not a location, or out of bounds} \\ \mathbb{Z} & \text{if } \mathbf{rpoint}_R(\mathtt{x}) \text{ or } \vec{m}^R_{*\mathtt{x}}(\vec{m}^R_{\mathtt{x}}(\mathtt{x})) \text{ is undefined} \\ \{\vec{m}^R_{*\mathtt{x}}(\vec{m}^R_{\mathtt{x}}(\mathtt{x}))\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_K(\vec{m}, R, \mathtt{x} \oplus \mathtt{y}) = \begin{cases} \mathbb{Z}, & \text{if } \vec{m}^R_{\mathtt{x}}(\mathtt{x}),\ \vec{m}^R_{\mathtt{y}}(\mathtt{y}), \text{ or } \vec{m}^R_{\mathtt{x}}(\mathtt{x}) \mathbin{\widetilde{\oplus}} \vec{m}^R_{\mathtt{y}}(\mathtt{y}) \text{ are undefined} \\ \{\vec{m}^R_{\mathtt{x}}(\mathtt{x}) \mathbin{\widetilde{\oplus}} \vec{m}^R_{\mathtt{y}}(\mathtt{y})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_K(\vec{m}, R, \&\mathtt{x}) = \{\mathbf{lval}(\mathtt{x})\}$$

$$\mathcal{E}_K(\vec{m}, R, \mathtt{x} \trianglelefteq \mathtt{y}) = \begin{cases} \{0, 1\}, & \text{if } \vec{m}^R_{\mathtt{x}}(\mathtt{x}),\ \vec{m}^R_{\mathtt{y}}(\mathtt{y}), \text{ or } \vec{m}^R_{\mathtt{x}}(\mathtt{x}) \mathrel{\widetilde{\trianglelefteq}} \vec{m}^R_{\mathtt{y}}(\mathtt{y}) \text{ are undefined} \\ \{\vec{m}^R_{\mathtt{x}}(\mathtt{x}) \mathrel{\widetilde{\trianglelefteq}} \vec{m}^R_{\mathtt{y}}(\mathtt{y})\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}_K(\vec{m}, R, p, \mathtt{x} \mathrel{\mathtt{:=}} E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } \mathcal{E}_K(\vec{m}, R, E) = \bot \\ \{(p, \vec{m}[\vec{m}^R_{\mathtt{x}} \mapsto \vec{m}^R_{\mathtt{x}}[\mathtt{x} \mapsto v]]) \mid v \in \mathcal{E}_K(\vec{m}, R, E)\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}_K(\vec{m}, R, p, {*}\mathtt{x} \mathrel{\mathtt{:=}} E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } \vec{m}^R_{\mathtt{x}}(\mathtt{x}) \text{ is undefined, not a location, or out of bounds;} \\ & \text{if } \mathbf{rpoint}_R(\mathtt{x}) \text{ is undefined; or if } \mathcal{E}_K(\vec{m}, R, E) = \bot \\ \{(p, \vec{m}[\vec{m}^R_{*\mathtt{x}} \mapsto \vec{m}^R_{*\mathtt{x}}[\vec{m}^R_{\mathtt{x}}(\mathtt{x}) \mapsto v]]) \mid v \in \mathcal{E}_K(\vec{m}, R, E)\}, & \text{otherwise.} \end{cases}$$

$$\mathbf{post}_K(\vec{m}, R, p, \mathtt{[}E\mathtt{]}) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \mathcal{E}_K(\vec{m}, R, E) = \bot \\ \emptyset, & \text{if } \mathcal{E}_K(\vec{m}, R, E) = \{0\} \\ \{(p, \vec{m})\}, & \text{otherwise.} \end{cases}$$

Figure 4.5: The interpretation of the partitioned analysis.

the sum of the concretizations defined by the separation environments in Q:

$$\gamma_{\widetilde{\mathcal{K}}\langle Q\rangle}(p,\vec{m}) = \bigcup_{(p,R)\in Q} \{(p,m) \mid m \in \gamma_R(\vec{m})\}$$

Lemma 4.1. *Let R be a separation environment, $\vec{m}$ a vector of memories, m a (single) memory, and $\mathtt{x}$ a variable. If m is in both $\gamma_{Sep}(R)$ and $\gamma_R(\vec{m})$ and $m(\mathtt{x})$ is an in-bounds location l, then $m(l)$ is equal to $\vec{m}^R_{*\mathtt{x}}(l)$.*

Proof. By the definition of γ_{Sep}, $\mathbf{region}_R(l)$ must be equal to $\mathbf{rpoint}_R(\mathtt{x})$. Hence, $\vec{m}^R_{*\mathtt{x}}$ is equal to $\vec{m}^R_l$. By the definition of γ_R, $m(l)$ must be equal to $\vec{m}^R_l(l)$. Hence, $m(l)$ is equal to $\vec{m}^R_{*\mathtt{x}}(l)$. $\square$

Lemma 4.2. *Let R be a separation environment, $\vec{m}$ a vector of memories, and m a (single) memory. If m is in both $\gamma_{Sep}(R)$ and $\gamma_R(\vec{m})$, then $\mathcal{E}_K(\vec{m}, R, E)$ equals $\mathcal{E}_C(m, E)$.*

Proof. We proceed by cases on E:

- $E = n$ or $E = \&\mathtt{x}$. These cases are trivial.

- $E = \mathtt{x}$, $E = \mathtt{x} \ \oplus \ \mathtt{y}$, or $E = \mathtt{x} \ \trianglelefteq \ \mathtt{y}$. By the definition of γ_R, $m(\mathtt{x})$ and $m(\mathtt{y})$ must be equal to $\vec{m}^R_{\mathtt{x}}(\mathtt{x})$ and $\vec{m}^R_{\mathtt{y}}(\mathtt{y})$, respectively. Hence, $\mathcal{E}_K(\vec{m}, R, E)$ will always be equal to $\mathcal{E}_C(m, E)$ in these cases.

- $E = *\mathtt{x}$. By the definition of γ_R, $m(\mathtt{x})$ must be equal to $\vec{m}^R_{\mathtt{x}}(\mathtt{x})$. If $m(\mathtt{x})$ is undefined, not a location, or out of bounds, then the property holds trivially.

  Assume that $m(\mathtt{x})$ is an in-bounds location l. By the definition of γ_{Sep}, $\mathbf{region}_R(l)$ must be equal to $\mathbf{rpoint}_R(\mathtt{x})$ (and, thus, $\mathbf{rpoint}_R(\mathtt{x})$ must be defined). It suffices to show that $\vec{m}^R_{*\mathtt{x}}(l)$ is equal to $m(l)$. This follows from Lemma 4.1.

This shows equality for $\mathcal{E}_K(\vec{m}, R, E)$ and $\mathcal{E}_C(m, E)$. $\square$

Lemma 4.3. *$\widetilde{\mathcal{I}}_{\mathcal{K}}\langle Q\rangle$ is sound for every set of separation states Q.*

Proof. Let $\mathcal{P}$ be a program and Q a set of separation states. Let $c = (en, m)$ be a concrete state in $\mathcal{I}_{\mathcal{C}}[\mathcal{P}]$. We must show that there is a state $(en, \vec{m})$ in $\widetilde{\mathcal{I}}_{\mathcal{K}}\langle Q\rangle$ that over-approximates c.

Let $\vec{m}$ be a vector of memories, with each component of $\vec{m}$ equal to m. Since m and $\vec{m}$ agree at all locations—in particular, all in-bounds locations—$(en, \vec{m})$ over-approximates c. Since the state c is in $\mathcal{I}_{\mathcal{C}}[\mathcal{P}]$, $m(l)$ is not a location, for all locations l. Hence, $\vec{m}^R_l(l)$ is not a location, for all locations l, and $(en, \vec{m})$ is in $\widetilde{\mathcal{I}}_{\mathcal{K}}\langle Q\rangle$. This shows soundness for $\widetilde{\mathcal{I}}_{\mathcal{K}}\langle Q\rangle$. $\square$

Lemma 4.4. *$\widetilde{\mathcal{F}}_{\mathcal{K}}\langle Q\rangle$ is $\gamma_{Sep}(Q)$-sound for every set of separation states Q.*

Proof. Let $\mathcal{P}$ be a program and Q a set of separation states. We must show $\widetilde{\mathcal{F}}_{\mathcal{K}}\langle Q\rangle[\mathcal{P}](p, \vec{m})$ over-approximates $\mathcal{F}_{\mathcal{C}}[\mathcal{P}](p, m)$ whenever (p, m) is in both $\gamma_{Sep}(Q)$ and $\gamma_{\widetilde{\mathcal{K}}\langle Q\rangle}(p, \vec{m})$.

Take concrete states $c = (p, m)$, $c' = (p', m')$, abstract state $a = (p, \vec{m})$, and statement S such that c is in both $\gamma_{\widetilde{\mathcal{K}}\langle Q\rangle}(p, \vec{m})$ and $\gamma_{Sep}(Q)$ and c' is an S-successor of c. Since Q is well-formed, there is exactly one separation state in Q for location p. Let R be the separation environment associated with that state. By definition, m is in both $\gamma_R(\vec{m})$ and $\gamma_{Sep}(R)$.

It suffices to show that there is some $\vec{m}'$ in $\widetilde{\mathcal{F}}_{\mathcal{K}}\langle Q\rangle[\mathcal{P}](\vec{m}, R, p', S)$ such that m' is in $\gamma_{\widetilde{\mathcal{K}}\langle Q\rangle}(\vec{m}')$. We proceed by cases on S:

- $S =$ `x := ` E. By Lemma 4.2, $\mathcal{E}(m, E)$ and $\mathcal{E}_K(\vec{m}, R, E)$ are equal. If they are both $\perp$, then the property is trivial. Assume they are not both $\perp$. Then,

$m' = m[\mathtt{x} \mapsto v]$ for some value v in $\mathcal{E}(m, E)$ and there exists some $\vec{m}'$ equal to $\vec{m}[\vec{m}^R_{\mathtt{x}} \mapsto \vec{m}^R_{\mathtt{x}}[\mathtt{x} \mapsto v]]$. By definition, m' is in $\gamma_{\widetilde{\mathcal{K}}\langle Q\rangle}(\vec{m}')$.

- $S = \mathtt{*x := } E$. Assume that $\vec{m}^R_{\mathtt{x}}(\mathtt{x})$ is an in-bounds location value l, $\mathbf{region}_R$ is defined for $\mathtt{x}$, and $\mathcal{E}_K(\vec{m}, R, E)$ is not equal to $\bot$; otherwise, the property is trivial. By the definition of γ_R, $m(\mathtt{x})$ must also be l. Since m is in $\gamma_{\mathit{Sep}}(R)$, we have also that $\mathbf{region}_R(l)$ is equal to $\mathbf{rpoint}_R(\mathtt{x})$.

  By the definition of $\mathbf{post}_C$, m' must be of the form $m[l \mapsto v]$, for some v in $\mathcal{E}_C(m, p, E)$. By Lemma 4.2, there exists some $\vec{m}'$ of the form $\vec{m}[\vec{m}^R_{\mathtt{*x}} \mapsto \vec{m}^R_{\mathtt{*x}}[l \mapsto v]]$ in $\widetilde{\mathcal{F}}_{\mathcal{K}}\langle Q\rangle[\mathcal{P}](\vec{m}, R, p', S)$. By definition, m' is in $\gamma_R(\vec{m}')$ if $m'(l')$ is equal to $\vec{m}'^R_{l'}(l')$ for all in-bounds locations l'. Since m is in $\gamma_R(\vec{m})$, it suffices to show that $\vec{m}'^R_{\mathtt{*x}}$ is equal to $\vec{m}'^R_{l}$. This is immediate, since $\mathbf{region}_R(l)$ is equal to $\mathbf{rpoint}_R(\mathtt{x})$.

- $S = \mathtt{[}E\mathtt{]}$. By Lemma 4.2, $\mathcal{E}(m, E)$ and $\mathcal{E}_K(\vec{m}, R, E)$ are equal. The property holds trivially.

This shows $\gamma_{\mathit{Sep}}(Q)$-soundness for $\widetilde{\mathcal{F}}_{\mathcal{K}}\langle Q\rangle$. □

Theorem 4.5. *The analysis generator $\widetilde{\mathcal{K}}$ is sound.*

Proof. By Definition 7, Theorem 1.4, Lemma 4.3, and Lemma 4.4. □

4.5 Case Study: Compressed Domain Names

To demonstrate the utility of our approach, we will describe a more complex application, taken from real code. We will show the definition of a real-world datatype, the annotations for a function operating on that datatype, and the results of using CASCADE to verify the function.

```
type Dn =
    label {
      tag:2 = 0b00,
      len:6 != 0b000000,
      name:u_char[len],
      rest:Dn
    }
  | indirect {
      tag:2 = 0b11,
      offset:14
    }
  | nullt {
      tag:8 = 0x00
    }
```

Figure 4.6: Definition of the `Dn` datatype.

A definition for the datatype `Dn`, representing an RFC 1035 *compressed domain name* [51], is given in Fig. 4.6. `Dn` is a union type with three variants: `label`, `indirect`, and `nullt`. The representation of a `label` value is: a 2-bit `tag` field (which must be zeroes), a 6-bit `len` field (which must *not* be all zeroes), a `name` field of exactly `len` bytes, and another `Dn` value `rest`, which follows immediately in memory. An `indirect` value has a 2-bit `tag` (which must be `0b11`) and a 14-bit `offset`. A `nullt` value has only an 8-bit `tag`, which must be zero. The constraints on the `tag` fields of `label`, `indirect`, and `nullt` allow us to distinguish between values.

Consider the function `ns_name_skip` in Fig. 4.7. The low-level pointer and bit-masking operations represent the traversal of the high-level `Dn` data structure. The correctness of the implementation is properly expressed in terms of that data structure.

In terms of the type `Dn`, the code in Fig. 4.7 is straightforward. The pointer `cp`, initialized with the value pointed to by the parameter `ptrptr`, points to a `Dn`

```
#define NS_CMPRSFLAGS (0xc0)

int
ns_name_skip(const u_char **ptrptr, const u_char *eom) {
    { allocated(*ptrptr, eom) }
    const u_char *cp;
    u_int n;

    cp = *ptrptr;
    { @invariant: cp ≤ eom ⟹
        cp + sizeOfDn(cp) = init(cp) + sizeOfDn(init(cp)) }
    while (cp < eom && (n = *cp++) != 0) {
        /* Check for indirection. */
        switch (n & NS_CMPRSFLGS) {
        case 0:               /* normal case, n == len */
            { isLabel(prev(cp)) }
            cp += n;
            { rest(prev(cp)) = toDn(cp) }
            continue;
        case NS_CMPRSFLGS:      /* indirection */
            { isIndirect(prev(cp)) }
            cp++;
            break;
        default:                /* illegal type */
            __set_errno (EMSGSIZE);
            return (-1);
        }
        break;
    }
    if (cp > eom) {
        __set_errno (EMSGSIZE);
        return (-1);
    }
    { cp = eom ∨ cp = init(cp) + sizeOfDn(init(cp)) }
    *ptrptr = cp;
    return (0);
}
```

Figure 4.7: The function `ns_name_skip` from BIND

value. The loop test (Line 12) assigns the first byte of the value to the variable `n` and advances `cp` by one byte. If `n` is `0`, then `cp` pointed to a `nullt` value and the loop exits. Otherwise (Line 14), the `switch` statement checks the two most significant bits of `n`—the `tag` field of a `label` or `indirect` value. If the `tag` field contains zeroes (Line 15), `cp` is advanced past the `label` field to point to the `Dn` value of the `rest` field. If the `tag` field contains ones (Line 20), `cp` is advanced past the `offset` field and breaks the loop. The `default` case of the `switch` statement returns an error code—the `tag` field was malformed. At the end of the loop, if `cp` has not exceeded the bound `eom`, the value of `cp` is one greater than the address of the last byte of the `Dn` value that `cp` pointed to initially. This is the contract of the function: given a reference to a pointer to a valid `Dn` value, it advances the pointer past the `Dn` value or to the bound `eom`, whichever comes first, and returns `0`; if the `Dn` value is invalid, it returns `-1`.

Annotating the source code. The datatype definition is translated into an inductive datatype with supporting functions and axioms, as in Section 4.2.2. The translation generates testers *isLabel*, *isIndirect*, and *isNullt*; selectors *len*, *name*, *rest*, etc.; and the encoding functions *toDn* and *sizeOfDn*. Each of these functions is now available for use in source code assertions, as in the bracketed, italicized portions in Fig. 4.7.

The annotations in Fig. 4.7 also make reference to some auxiliary functions: `init(x)` represents the initial value of a variable `x` in the function; `prev(x)` represents the previous value of a variable `x` in a loop (i.e., the value at the beginning of an iteration).

On entry to the function (Line 5), we assume that the region pointed to by `*ptrptr` and bounded by `eom` is properly allocated. To each `switch` case (Lines

15 and 20), we add an assertion stating that the observed `tag` value (i.e, `n & NS_CMPRSFLGS`) is consistent with a particular datatype constructor (i.e., `label` or `indirect`). (Note that `prev(cp)` refers to the value of `cp` *before* the loop test, which has side effects). The loop invariant (Lines 10-11) states that `cp` advances through the `Dn` data structure pointed to by `init(cp)`—in each iteration of the loop, if `cp` has not exceeded the bound `eom`, it points to a `Dn` structure (perhaps the "tail" of a larger, inductive value) that is co-terminal with the structure pointed to by `init(cp)`. On termination, the loop invariant implies the desired post-condition: if no error condition has occurred, `*ptrptr` will point to the byte immediately following the `Dn` value pointed to by `init(cp)`—the pointer will have "skipped" the value. Note that we do not require an assertion stating that `cp` is reachable from `init(cp)` via `rest` "pointers" to prove the desired property—the property is provable using purely inductive reasoning.

Using the code annotations, Cascade can verify the function by generating a set of verification conditions representing non-looping static paths through the function. Fig. 4.8 gives an example of such a verification condition. It represents the path from the head of the loop through the `0` case of the `switch` statement (Line 15), ending with the `continue` statement (Line 19) and asserting the preservation of the loop invariant. (Note that we assume here that pointers are 8 bits. Larger pointer values are easily handled, but the formulas are more complicated.) As in Section 4.3, the verification condition uses a partitioned memory model with two memory arrays, s and h: the values of local variables and parameters are stored in s while the `Dn` value pointed to by `cp` is stored in h. Proposition (4.1) asserts the loop invariant on entry. Propositions (4.2)–(4.5) represent the evaluation, including effects, of the loop test. Proposition (4.6) represents the matching of the

$$
\begin{array}{cr}
s_0[\texttt{\&cp}] > s_0[\texttt{\&eom}] & \\
\lor\ s_0[\texttt{\&cp}] + \mathit{sizeOfDn}(\mathit{toDn}(h_0, s_0[\texttt{\&cp}])) & \\
= \mathit{init}(\texttt{\&cp}) + \mathit{sizeOfDn}(\mathit{toDn}(h_0, \mathit{init}(\texttt{\&cp}))) & (4.1) \\
s_0[\texttt{\&cp}] < s_0[\texttt{\&eom}] & (4.2) \\
s_1 = s_0[\texttt{\&cp} \mapsto s_0[\texttt{\&cp}] + 1] & (4.3) \\
s_2 = s_1[\texttt{\&n} \mapsto h_0[s_0[\texttt{\&cp}]]] & (4.4) \\
s_2[\texttt{\&n}] \neq 0 & (4.5) \\
s_2[\texttt{\&n}][7:6] = 0 & (4.6) \\
\mathit{is\_label}(\mathit{toDn}(h_0, s_0[\texttt{\&cp}])) & (4.7) \\
s_3 = s_2[\texttt{\&cp} \mapsto s_2[\texttt{\&cp}] + s_2[\texttt{\&n}]] & (4.8) \\
\mathit{rest}(\mathit{toDn}(h_0, s_0[\texttt{\&cp}])) = \mathit{toDn}(h_0, s_3[\texttt{\&cp}]) & (4.9) \\
\hline
s_4[\texttt{\&cp}] > s_4[\texttt{\&eom}] & \\
\lor s_4[\texttt{\&cp}] + \mathit{sizeOfDn}(\mathit{toDn}(h_0, s_4[\texttt{\&cp}])) & \\
= \mathit{init}(\texttt{\&cp}) + \mathit{sizeOfDn}(\mathit{toDn}(h_0, \mathit{init}(\texttt{\&cp}))) & (4.10)
\end{array}
$$

Figure 4.8: The verification condition for preservation of the loop invariant in the `0` case of `ns_name_skip`.

`switch` case. Propositions (4.7)–(4.9) capture the body of the case block. Finally, Proposition (4.10) (the proposition we would like to prove, given the previous assumptions) asserts the preservation of the loop invariant.

4.5.1 Experiments

Table 4.1 shows the time taken by Cvc3 to prove the verification conditions generated by Cascade for `ns_name_skip`, using both the flat and partitioned memory models. The times given are for a Intel Dual Core laptop running at 2.2GHz with 4GB RAM and do not include the time needed for separation analysis or verification condition generation (which is trivial). Each VC represents a non-looping, non-erroneous path to an assertion. The two Term VCs represent the loop exit paths: Term (1) is the path where the first conjunct is false (`cp >= eom`); Term

Table 4.1: Running times on `ns_name_skip` VCs.

Name	Lines	Time (seconds) Flat	Part.
INIT	5–12	0.34	0.03
CASE `0` (1)	12-16	13.94	0.05
CASE `0` (2)	12-28	33.42	0.06
CASE `0` (3)	12-19	*	0.12
CASE `0xc0` (1)	12–14, 20–21	6.14	0.04
CASE `0xc0` (2)	12–14, 20–23, 30, 34	*	0.07
TERM (1)	12, 30, 34	0.63	0.06
TERM (2)	12, 30, 34	*	0.05

(2) is the path where the first conjunct is true (`cp < eom`) and the second is false (`n == 0`). The verification conditions marked with * for the flat memory model timed out after two minutes—we believe that these formulas are not provable in CVC3 (indeed, they may not be valid in the flat model). All of the verification conditions together can be validated using the partitioned memory model in less than one second.

4.6 Related Work

Some early work on verification of programs operating on complex datatypes was done by Burstall [12], Laventhal [44], and Oppen and Cook [55]. Their work assumes that data layout is an implementation detail that can be abstracted away. Our work here focuses on network packet processing code, where the linear layout of the data structure is an essential property of the implementation.

More recently, O'Hearn, Reynolds, and Yang [54] have approached the problem using *separation logic* [60, 34]. Given assumptions about the structure of the heap, the logic allows for powerful localized reasoning. In this work, we use separation analysis in the style of Hubert and Marché [33] and Rakamarić and Hu [59] to establish separation invariants, thus “localizing” the verification conditions.

Conclusions

The work in this thesis grew out of a simple desire to incrementally improve the state of the art in program verification. In order to do so, I first had to solve several preliminary problems.

First came the question of soundness. In developing a verification tool for C code, I soon came to realize I would need pointer analysis to make the tool effective. In reviewing the literature on pointer analysis, I was frustrated to find the soundness claims vague and imprecise. After careful study, I was able to describe precisely the conditions under which a typical pointer analysis would be sound, but this notion of conditional soundness did not correspond to any notion of soundness commonly used in the static analysis community. This led to the development of the framework presented in Chapter 1. In this thesis, I have used the conditional soundness framework to precisely describe pointer analysis combined with memory safety analysis (Chapter 2) and a memory partitioning analysis combined with datatype analysis (Chapter 4). But the framework is by no means restricted only to these domains. Many, if not most, program analyses are sound only under certain assumptions about program behavior—for example, many analyses assume the program is sequentially consistent, that integer overflow does not occur, or that the program is free of floating point exceptions. The soundness claims of such analyses could be refined using the conditional soundness framework I have described—it would be of significant benefit to the community if they were.

Next came the issue of tool support. To advance the research and educational goals of the Analysis of Computer Systems research group, Clark Barrett and Amir Pnueli saw the need for a flexible, powerful, open source verification platform with a state-of-the-art SMT solver back-end. With the help of other members of ACSys, most especially Morgan Deters and Dejan Jovanović, I led the development of CASCADE to meet this need. The result is a software framework that enabled me to pursue the research described in Chapter 4. I hope it will prove to be as useful to future students and researchers.

With soundness results and tool support in hand, I began to experiment with adding high-level datatype assertions to C code. The approach I took depended crucially on the features of CVC3—in particular, being able to the combine inductive datatypes, bit vectors, arrays, and uninterpreted functions in a single formula. Although the examples in Chapter 4 are modest, I believe this technique can scale to several hundreds or thousands of lines of code. This research shows there are real benefits to utilizing the full expressive power of SMT solvers in verification.

We are still a long way from push-button verification tools that can guarantee that rockets, energy management systems, or medical devices will never fail in harmful and costly ways. Indeed, tools of such power are a practical impossibility. However, every day progress is being made in improving software quality using formal methods. It is my hope that the work described in this thesis represents some small contribution to that progress.

Bibliography

[1] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, 1993.

[2] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *Static Analysis Symposium (SAS)*, volume 2477 of *Lecture Notes in Computer Science*, pages 230–246, 2002.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[4] American National Standard for Programming Languages - C, August 1989. ANSI/ISO 9899-1990.

[5] Sara Baase. *A Gift of Fire: Social, Legal, and Ethical Issues for Computing and the Internet*. Prentice Hall, 2007.

[6] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods (IFM)*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20, 2004.

[7] Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, 2007. Berlin, Germany.

[8] Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomas Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, 1996.

[9] Nicolas Bourbaki. Sur le théorème de Zorn. *Archiv der Mathematik*, 2(6):434–437, 1949.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442, 2006.

[11] Glenn Bruns and Satish Chandra. Searching for points-to analysis. In *Foundations of Software Engineering (FSE)*, pages 61–70, 2002.

[12] R.M Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence*, volume 7, 1972.

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

[14] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

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

[16] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming (ESOP)*, pages 21–30, 2005.

[17] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the ASTRÉE static analyzer. In *Asian Computing Science Conference (ASIAN)*, pages 272–300, 2006.

[18] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation (PLDI)*, pages 35–46, 2000.

[19] David Delmas and Jean Souyris. ASTRÉE: from research to industry. In *Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451, 2007.

[20] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *Programming Language Design and Implementation (PLDI)*, pages 144–157, 2006.

[21] Nurit Dor. *Automatic Verification of Program Cleanness.* PhD thesis, Tel Aviv University, December 2003.

[22] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Programming Language Design and Implementation (PLDI)*, pages 155–167, 2003.

[23] Mark Dowson. The Ariane 5 software failure. *Software Engineering Notes*, 22(2):84, 1997.

[24] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[25] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, 2007.

[26] U.S.-Canada Power System Outage Task Force. Final report on the August 14, 2003 blackout in the United States and Canada: Causes and recommendations, April 2004.

[27] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report UCB/CSD-97-964, University of California, Berkeley, August 1997.

[28] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in Satisfiability Modulo Theories. In *Computer Aided Verification (CAV)*, pages 306–320, 2009.

[29] Robert Grimm. Better extensibility through modular syntax. In *Programming Language Design and Implementation (PLDI)*, pages 38–51, 2006.

[30] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *Programming Language Design and Implementation (PLDI)*, 2006.

[31] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Programming Language Design and Implementation (PLDI)*, pages 24–34, 2001.

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

[33] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV)*, pages 81–93, 2007.

[34] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. *SIGPLAN Notices*, 36(3):14–26, 2001.

[35] Programming Language C++, August 2003. ISO/IEC 14882:2003.

[36] ISO Standard - Programming Languages - C, December 1999. ISO/IEC 9899:1999.

[37] Akihiro Kanamori. The mathematical import of Zermelo's well-ordering theorem. *The Bulletin of Symbolic Logic*, 3(3), 1997.

[38] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. *Computer Communication Review*, 29(4):3–13, 1999.

[39] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[40] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Principles of Programming Languages (POPL)*, pages 93–103, 1991.

[41] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Programming Language Design and Implementation (PLDI)*, pages 235–248, 1992.

[42] Jean-Louis Lassez, V. L. Nguyen, and Liz Sonenberg. Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

[43] C. Lattner. *Macroscopic Data Structure Analysis and Optimization.* PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[44] Mark S. Laventhal. Verifying programs which operate on data structures. In *Reliable Software*, pages 420–426, 1975.

[45] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Principles of Programming Languages (POPL)*, pages 270–282, 2002.

[46] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. Melange: creating a "functional" internet. In *European Conference on Computer Systems (EuroSys)*, pages 101–114, 2007.

[47] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, New York, NY, USA, 1992.

[48] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, New York, NY, USA, 1995.

[49] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[50] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 54–63, 2006.

[51] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. `http://www.ietf.org/rfc/rfc1035.txt`.

[52] Yannick Moy and Claude Marché. Jessie. `http://frama-c.cea.fr/jessie.html`.

[53] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, pages 128–139, 2002.

[54] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic (CSL)*, pages 1–19, 2001.

[55] Derek C. Oppen and Stephen A. Cook. Proving assertions about programs that manipulate data structures. In *Symposium on the Theory of Computing (STOC)*, pages 107–116, 1975.

[56] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer Aided*

Verification (CAV), volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, 1996.

[57] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: a yacc for writing application protocol parsers. In *Internet Measurement Conference (IMC)*, pages 289–300, 2006.

[58] Amir Pnueli and Elad Shahar. A platform for combining deductive with algorithmic verification. In *Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 184–195, 1996.

[59] Zvonimir Rakamarić and Alan J. Hu. A scalable memory model for low-level code. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 290–304, 2009.

[60] J. C Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, pages 303–321, 2000.

[61] Roger Sessions. The IT complexity crisis: Danger and opportunity. White paper, November 2009.

[62] Nikhil Sethi and Clark Barrett. CASCADE: C assertion checker and deductive engine. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 166–169, 2006.

[63] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[64] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[65] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[66] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, 1995.

[67] Extensible markup language (XML) 1.0 (Fifth edition). W3C Recommendation, November 2008. `http://www.w3.org/TR/xml/`.