Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
Software Languages Lab

Purity Analysis for Higher-Order Imperative Languages: An Abstract Machine Approach

Dissertation submitted for the Degree of Doctor of Philosophy in Sciences

Jens Nicolay

Promotors: Prof. Dr. Wolfgang De Meuter
Prof. Dr. Coen De Roover

May 2016

Abstract

Side effects are an essential part of many programs. However, side effects make it harder to understand program behavior, because expressions no longer can be treated as mere values. Pure expressions have no side effects, and research in different areas has demonstrated that purity aids program specification, optimization, testing, debugging, and maintenance.

Determining side effects is useful for program comprehension and optimization, but it is a difficult problem and particularly so in the presence of objects and higher-order procedures that may flow freely through a program. In practice it is not sufficient to detect the presence of side-effecting expressions in a program. The dynamic extent of the side effect must also be accurately established to obtain useful information. A side effect in one particular context, e.g. a procedure application, may not be observable outside that context. In this regard, manual or crude approaches to establish the extent of side effects are too imprecise and error-prone.

This dissertation explores and evaluates techniques for statically computing the side-effecting behavior of higher-order imperative programs to determine procedure purity.

We base our approach on an abstract state machine that is an interpreter for a core Scheme language, instrumented to register read and write effects on resources. Resources in our semantics are variables and objects, which are allocated in a store at a specific address. Following the AAM approach, the machine is parameterized to be able to express both concrete and abstract semantics. The result of program interpretation is a flow graph that is consumed by client analyses interested in program properties involving control flow, value flow, and effects.

Our first contribution is a procedure side-effect analysis that computes for each procedure application the side effects that are observable by direct and indirect callers. Applications and associated callers are found by traversing all reachable application contexts on the call stack at the point where an effect occurs. Observability of effects is based on freshness of resources. A

resource is fresh in a particular context if it was created in that context. Effects on fresh resources can never be observed outside that context. For example, if during procedure application an object is allocated, then that object is fresh in that application context, and any effects on it can never be observed by callers. We discuss three characterizations of freshness. The first, address freshness, is attractive because it is a close fit with the store-semantics of the abstract machine. However, this is not always ideal because termination of static analysis is primarily guaranteed by allocating resources at addresses that are already in use, and therefore not fresh. For example, a variable is typically allocated at the same address throughout an entire abstract interpretation. To improve the precision of side-effect analysis, we introduce two additional scope-based characterizations of freshness. Variable freshness is based on locality of variables, i.e. whether they are local or free with respect to a procedure's scope, and object freshness keeps track of the flow of objects in and out of scopes through object references. Their formalization forms the second contribution of this work.

Our third and final contribution is the design of a purity analysis on top of procedure side-effect analysis. Purity analysis classifies procedures as either pure, observer, or procedure. A procedure is pure if none of its applications generate or depend upon externally observable side effects. A procedure is an observer as soon as one of its applications depends on an external side effect, but none of its applications generate observable side effects. Otherwise, a procedure is classified as a procedure.

We apply the analyses presented in this work to a set of programs, and discuss the outcome in terms of four key aspects: correctness, soundness, precision, and performance. We find that our purity analysis is capable of uncovering purity in a variety of programs. Also in this setting our experiments show that our analysis is capable of correctly classifying functions in terms of the side effects they generate and depend upon.

Samenvatting

Neveneffecten zijn een essentieel onderdeel van veel programma's. Toch maken ze het moeilijker om het gedrag van programma's te voorspellen, omdat expressies niet langer enkel een waarde voorstellen. Pure expressies hebben geen neveneffecten, en onderzoek in verschillende domeinen heeft aangetoond dat puurheid helpt bij de specificatie, optimalisatie, en het testen, debuggen, en onderhoud van programma's.

Het bepalen van neveneffecten is nuttig voor het begrijpen en optimaliseren van programma's, maar is erg moeilijk, zeker in aanwezigheid van objecten en hogere-orde procedures die vrij doorheen een programma mogen vloeien. Daarenboven volstaat het niet om enkel de aanwezigheid van expressies met neveneffecten in een programma aan te tonen. Ook de dynamische reikwijdte van elk neveneffect moet zo precies mogelijk worden vastgesteld om bruikbare informatie te bekomen. Een neveneffect in een bepaalde context, bijvoorbeeld de toepassing van een procedure, is misschien niet waarneembaar buiten die context. Manuele of ruwe benaderingen om de reikwijdte van neveneffecten vast te stellen zijn te onprecies en erg foutgevoelig.

Deze verhandeling onderzoekt en evalueert technieken om het optreden van neveneffecten in een programma statisch te berekenen, om zo puurheid van procedures te bepalen. We baseren onze aanpak op een abstracte state machine die een interpreter van een basis Scheme taal voorstelt, en geïnstrumenteerd is om lees- en schrijfeffecten van en naar resources te registreren. Resources in onze semantiek zijn variabelen en objecten die worden gealloceerd op bepaalde adressen in een store. We volgen de AAM techniek door de machine te parameterizeren zodanig dat deze zowel abstracte als concrete semantiek kan uitdrukken. Het resultaat van programma-interpretatie is een graaf die geconsumeerd kan worden door client-analyses geïnteresseerd in programma-eigenschappen omtrent control flow, value flow, en effecten.

Onze eerste bijdrage is een analyse voor het berekenen van procedure neveneffecten die voor elke procedure-toepassing de neveneffecten berekent die waarneembaar zijn door directe en indirecte oproepers. Toepassingen en bijbehorende oproepers worden gevonden door alle bereikbare toepassing-contexten op de stack te overlopen op het punt waar een effect zich voordoet. De waarneembaarheid van een effect is gebaseerd op de versheid van de betrokken resource. Een resource is vers in een bepaalde context als deze werd gecreëerd in deze context. Effecten op verse resources kunnen nooit waargenomen worden buiten deze context. Bijvoorbeeld, als tijdens een procedure-toepassing een object wordt gealloceerd, dan is dat object vers in de context van die toepassing, en alle effecten op de resource kunnen nooit waargenomen worden door oproepers. We bespreken drie karakterisaties van versheid. De eerste, versheid van adressen, is attractief omdat het nauw aansluit met de store-semantiek van onze abstracte machine. Maar versheid bepalen aan de hand van adressen is niet altijd ideaal omdat de eindigheid van statische analyse vooral gegarandeerd wordt door het toestaan van resource-allocatie op adressen die reeds in gebruik zijn, en daarom niet vers. Bijvoorbeeld, variabelen worden typisch op hetzelfde adres gealloceerd doorheen een abstracte interpretatie van een programma. Om de precisie van de analyse van neveneffecten te verbeteren, introduceren we twee bijkomende karakterisaties van versheid gebaseerd op scope. Versheid van variabelen is gebaseerd op de localiteit van variabelen, d.w.z. of ze vrij of gebonden zijn in de scope van een procedure, en versheid van objecten wordt bijgehouden door te kijken hoe ze worden doorgegeven tussen verschillende scopes en object-referenties. Hun formalisatie vormt de tweede bijdrage van dit werk.

Onze derde en laatste bijdrage is het ontwerp van een puurheidsanalyse bovenop de neveneffecten-analyse voor procedures. De puurheidsanalyse classifieert procedures als zijnde puur, observer, of simpelweg procedure. Een procedure is puur als geen van haar toepassingen een waarneembaar effect genereert of afhankelijk is van externe neveneffecten. Een procedure is een observer van zodra een van haar toepassingen afhankelijk is van een extern neveneffect, maar geen enkele een waarneembaar neveneffect genereert. Anders wordt de procedure beschouwd als een gewone procedure.

We passen de analyses besproken in dit werk toe op een verzameling programma's en bespreken het resultaat in termen van vier belangrijke criteria: correctheid, deugdelijkheid, precisie, en performantie. We concluderen dat onze puurheidsanalyse in staat is om puurheid bloot te leggen in een diverse verzameling programma's. We hebben onze aanpak ook toegepast voor het detecteren van pure functies in JavaScript. Ook in dit kader tonen

onze experimenten aan dat onze analyse in staat is om correct functies te classifiëren in termen van neveneffecten die ze genereren of waarvan ze afhankelijk zijn.

Acknowledgements

I would like to start by thanking the members of my jury (Anders Møller, Frank Piessens, Ann Nowé, Beat Signer, Ann Dooms, Theo D'Hondt), for their insightful comments and feedback.

Thanks to IWT for providing me with four years of funding to complete this work.

Put your hands together for my promotors! (How many times can one write "thanks to..."?) Coen, for all the guidance and support over the years, which of course merits a lot more than this one crappy sentence (although I did mention you first). Wolfgang, for providing me the opportunity to start a PhD, and for reminding me of the big picture when I was lost in a swamp of details.

I also want to sincerely thank my other colleagues at SOFT. Theo, for sparking my interest in computer science in general, and interpreters in particular. Quentin, for all the help with my implementation and experiments, and for listening to my ideas-du-jour. Angela, for the kind words of support when I was tired of... well, just tired. Carlos, for all the "fascinating" discussions in our office. Elisa, for bringing cookies and chocolate, and sometimes hiding them in places were I was able to find them. Janwillem, for being a better candy shop manager than I was (although your Loyalty Card Extreme free cookies are making me fat). Laure, for the #random bat pictures and #general cheerful contributions on #earlybirds. Mattias, member of the IWT triangle, for all the crazy talk, music, and sand. Nathalie, for literally punching holes through my dissertation, and for liking the Efteling and therefore transitively me. Niels, for the politically correct lunchtime discussions. Reinout, member of the IWT triangle, for being part of the living proof that not only our national soccer team is a golden generation. Simon, for harassing me with questions about cola, and for the etymological quests on his yellow(!) phone. Stefan, for the politically incorrect lunchtime discussions. Thierry, for his sausage screen. All the other colleagues at SOFT, current and former: thanks for making SOFT a fun place to work.

Apology to Dirk for bricking Wilma with a fork-bomb.

During my endeavours, the Mottige Dassen arranged for some distractions in the form of Halloween weekends, BBQs, and the like.

As always, my family helped me tremendously, before and while doing my PhD, and I cannot thank them enough for this. My mother, for inspiring me to read and learn. My father, for buying my first computer (a Commodore 64). Mima, Pipa, Bonni, Bompa piraat, and all other family members who helped look after the kids and the house while I was working against yet another deadline.

A big thanks and hug for the Monkey, for being there for me and the children, even when I was not there for you (sorry).

For Sloeber, Sloebina, and Rover: if you set your mind to it, you can do it!

Contents

Chapter 1

Introduction

This dissertation presents and evaluates techniques for statically computing the side-effecting behavior of higher-order imperative programs to determine procedure purity.

1.1 Research Context

Most of the popular contemporary programming languages are imperative. For example, among the more than 2 million open-source projects hosted on repository site `github.com`, JavaScript is used most frequently. It is followed by Java, Python, PHP, and Ruby—all of which are imperative programming languages.

The imperative programming paradigm is characterized by the execution of ordered sequences of instructions that modify program state through side effects. Imperative programs can be difficult to reason about, as reasoning about an individual instruction requires considering the entire execution history of the program [Backus, 1978].

Even though popular contemporary languages are imperative, these languages are increasingly featuring concepts from the functional programming paradigm, either by design or through evolution.

The functional programming paradigm is characterized by higher-order programming without side effects. Higher-order programming improves the expressivity and modularity of programs [Abelson and Sussman, 1983, Hughes, 1989]. The absence of side effects, a property called purity, facilitates reasoning about the behavior of functional programs.

Languages such as JavaScript, Python, and Ruby, have adopted higher-order programming and its advantages. However, unlike functional programs,

imperative programs in these languages still contain side effects. Higher-order imperative programs therefore cannot take full advantage of the benefits that purity can bring, such as a more elegant and solid foundation for program comprehension and automated verification.

To enable higher-order imperative programs to profit more fully from the advantages of functional programming, we propose a static analysis for detecting side effects and for determining their dynamic extent as precisely as possible. The results of the analysis enable reasoning in a more functional way over larger parts of an imperative program.

Static analysis of higher-order imperative programs is challenging. However, recent advances in static analysis have made it possible to analyze programs containing imperative and functional features with sufficient precision [Gilray et al., 2016, Johnson and Van Horn, 2014]. In this dissertation, we build upon these recent advances, and present techniques for statically computing the side-effecting behavior of higher-order imperative programs with the goal of determining procedure purity.

Side Effects

Expressions evaluate to values, and everything else an expression does is called a side effect of that expression. Side effects are used to efficiently update data structures or perform I/O, and therefore an essential part of many programs. However, side effects make it harder to understand program behavior, because expressions can no longer be treated as only the values they evaluate to. Instead, it becomes more difficult to reason about the interaction between expressions, and the program evaluation history has to be taken into account when determining the outcome of evaluating expressions.

Purity

Pure expressions have no side effects. Research in different areas has demonstrated that purity aids program specification, optimization, testing, debugging, and maintenance [Finifter et al., 2008]. Purity facilitates establishing security and confidentiality guarantees about applications. It also has the potential to reduce the number of bugs in programs, and makes it easier to reproduce bugs. Pure procedures can be more safely called from assertions and contracts. It is often necessary to enforce purity in order to ensure modularity, and compilers and runtimes can use purity to optimize program

performance. In summary, detection or verification of purity is useful for a variety of software engineering purposes.

The absence of side effects is in general not sufficient for expressions and procedures to be regarded as safe and deterministic. Other factors, like non-termination and resource starvation also play a role, but are out of the scope of this dissertation.

Extent of Side Effects

In practice it is not sufficient to only detect the presence of side-effecting expressions in a program. To effectively make use of side effect information, it is as important to determine where in the program a side effect can actually be observed. Consider for example a library for creating and manipulating immutable data structures. When performing a batch insert, the library may decide that it is more beneficial to transform an immutable data structure into a mutable copy. The mutable copy can then be more efficiently updated using side effects, instead of applying a succession of functional updates on the immutable original. After updating, the updated mutable copy is converted into an immutable data structure and returned to the caller. Although clearly side effects are involved in this scenario, the user of the library cannot observe these effects and can reason about the library as being side-effect free.

Therefore, instead of determining whether side effects occurs, it is often more interesting to know *where* in a program side effects occur, and to which *extent* they have consequences. Constraining side effects as tightly as possible makes it less difficult to reason about a program as a whole.

Procedure Side Effects

Procedures are at an interesting level of code granularity to determine side effects. Procedure applications establish contexts in a program that enable us to distinguish between effects that are local and therefore not observable outside that context, and all other effects that are observable outside that context. Additionally, procedures are units of code that often are executed many times during program execution, and they can be named, exported, and memoized. For these reasons it is interesting to treat side effects at the level of procedures and procedure calls. A procedure side effect then is a effect that is observable during at least one application of that procedure.

1.2 Problem Statement

Information about side effects and their extent is useful for program comprehension and optimization, but it is difficult to obtain this information statically, and particularly so in the presence of objects and higher-order procedures that may flow freely through a program.

While side effects may be straightforward to determine for most individual expressions in isolation, procedure calls complicate matters. Obviously, it is possible to do a crude approximation of procedure side effects that occur during program evaluation. For example, one could look for syntactic expressions that mutate a variable or object, and then mark the enclosing procedure as having observable side effects. However, procedures can call other procedures. To determine the extent of the side effect, it is necessary to inspect all procedures that directly and indirectly call the side-effecting procedure to examine whether the side effect is observable at call sites. Following the same principle, if the examined procedure calls other procedures, which in turn call other procedures, the effects of those procedures also have to be taken into account.

Even if we can compute calling information, then there is still the problem of objects that are created and manipulated during procedure application. Objects can be created during an application, passed to procedures as parameters, referenced through a free variable, or retrieved from a data structure. Variables, although governed by lexical scoping, can be non-local (or free) in certain procedures. If the procedure in which a variable is free behaves like a value, then the free variable can be considered to be similar to an object field.

In summary, determining side effects and their extents requires detailed knowledge of the control flow and value flow of a program. It is not feasible nor desirable to obtain this information by naive approximations that offer no useful precision.

1.3 Thesis

Procedure purity can be effectively approximated statically in a higher-order, imperative program by performing an abstract interpretation of that program to detect side effects and determine their extent through stack reachability.

This dissertation supports the thesis by presenting and evaluating techniques for statically computing the side-effecting behavior of higher-order imperative programs to determine procedure purity.

1.4 Approach

We base our approach on an abstract state machine that is an interpreter for a core Scheme language SCHEME$_0$. We chose Scheme because it is an elegant functional language with support for imperative programming. Following the AAM approach (Abstracting Abstract Machines) [Van Horn and Might, 2010], the machine is abstracted using abstract interpretation, and parameterized so that it can be configured to express either concrete or abstract semantics. Abstract interpretation is a good match for static analysis, because it provides a theory to develop static analyses that are correct and provide meaningful results about the runtime behavior of programs. The resulting machine interpreter is also instrumented to register read and write effects on resources, which in the context of this work we consider to be variables and objects. The result of program interpretation is a flow graph that is consumed by client analyses interested in program properties that involve control flow, value flow, and effects.

Resources are stored at specific memory locations, and this memory is modeled as a store (essentially a heap) mapping addresses to values. Reading from and writing to memory locations by expressions are considered to be effects. Procedure side effects are latent in nature, and are therefore linked to the behavior of applications of a particular procedure at runtime, which can be obtained from a program's flow graph. A procedure side effect is observable when the effect is on a resource that is reachable by a caller of that procedure. When it can be determined that a side effect is unobservable in a certain context, we say that the effect can be masked. Our notion of procedure purity allows unobservable side effects, therefore allowing procedures that allocate and mutate memory locations or depend on mutable memory locations to still be considered free from side effects. To increase the usefulness of side-effect analysis, its goal is to soundly mask as many effects as possible.

Determining the observable extent of side effects in terms of application contexts has to happen for all procedures that are in the process of computing a result when an effect occurs [Might and Prabhu, 2009]. The set of active application contexts can be obtained by traversing the call stack. For every

effect that occurs, the observability of that effect will be examined in the context of every active application context. If the effect is not observable in a particular application context, then all application contexts that are reachable from that context will also not observe the effect.

1.5 Contributions

This dissertation presents a framework for side-effect and purity analysis and purity analysis for higher-order imperative programs using abstract machines. The framework is modular because we present several analyses that operate over a flow analysis and work together.

Our first contribution is a procedure side-effect analysis that computes for each procedure application the side effects that are observable by direct and indirect callers. Observability of effects is based on freshness of resources. A resource is fresh in a particular context if it was created in that context, i.e., it did not exist before. Effects on fresh resources can never be observed outside that context. For example, if during procedure application an object is allocated, then that object is fresh in that application context, and any effects on it can never be observed by callers. We discuss three characterizations of freshness. The first, address freshness, is attractive because it is a close fit with the semantics of the abstract machine in which resources are mapped in the store by addresses. However, address freshness is not always ideal because termination of static analysis is primarily guaranteed by allowing the analysis to allocate resources at addresses that are already in use, and therefore not fresh. For example, a variable is typically allocated at the same address throughout an entire abstract interpretation.

To improve the precision of side-effect analysis in an abstract setting, we introduce two additional scope-based characterizations of freshness. Variable freshness is based on locality of variables, i.e., whether they are local or free with respect to a procedure's scope. Object freshness keeps track of the flow of objects in and out of scopes through object references. The formalization of freshness analysis for variables and objects forms the second contribution of this work.

Our third and final contribution is the design of a purity analysis on top of procedure side-effect analysis. Our purity analysis classifies procedures as either **pure**, **observer**, or **procedure**. A procedure is **pure** if none of its applications generate or depend upon externally observable side effects. A procedure is an **observer** as soon as one of its applications depends on

an external side effect, but none of its applications generate observable side effects. Otherwise, a procedure is classified as a **procedure**.

We summarize the main contributions this dissertation makes.

- A procedure side-effect analysis that computes observable side effects for each procedure application (Chapter 3).
- A freshness analysis for addresses, variables, and objects (Chapter 4), coupled with escape analysis for increased precision (Chapter 5).
- A purity analysis for higher-order, imperative programs (Chapter 6).

In addition to the main contributions, we also perform an extensive evaluation of the analyses presented in this dissertation (Chapter 8). We apply the analyses to a set of benchmarks, and discuss the outcome in terms of four key aspects: correctness, soundness, precision, and performance. We find that our purity analysis is capable of detecting purity in a variety of programs.

Finally, we transpose our approach for detecting procedure purity for Scheme to JavaScript (Chapter 9). Like Scheme, JavaScript has both a functional and an imperative core, but the language also supports object-oriented programming with prototypal inheritance. JavaScript is a dynamic and complex language that makes static analysis challenging. Porting our approach to JavaScript therefore enbles us to evaluate how our ideas and techniques can be realized in this more dynamic and complex setting.

1.6 Dissertation Outline

This dissertation is structured as follows.

Chapter 2: Syntax and Semantics of SCHEME$_0$

We begin our discourse with a discussion of the features of *higher-order imperative languages*. We then introduce our core Scheme input language SCHEME$_0$, which we use throughout most of this dissertation. To model the semantics of SCHEME$_0$, we formalize an *abstract state machine* that is an interpreter for SCHEME$_0$. The result of program interpretation is a *flow graph* representing runtime program behavior, which can be consumed by client analyses interested in program properties involving control flow and value flow. The abstract machine can be configured to express either concrete or abstract semantics.

With *concrete semantics*, the graph has full precision but can become unwieldingly large, or even infinite. With finite *abstract semantics*, the graph becomes finite, although as a result it may overapproximate program behavior.

Chapter 3: Side-Effect Analysis

The side-effect analysis presented in this chapter is the first main contribution of this dissertation. We instrument the abstract machine from the previous chapter to register *read and write effects* on variables and object properties. These effects are also reflected in the flow graph, enabling client analyses to reason about effects. Procedure side-effect analysis computes for each procedure application the side effects that are *observable* by direct and indirect callers. The analysis does so by considering all effects in a flow graph, and for every effect it traverses all *reachable application contexts* on the call stack at the point where the effect occurs. If an effect targets a resource that is reachable in the store of the caller in a particular application context, then the effect is observable in that context; else the effect can be *masked* by the analysis. We call this form of observability address-based observability. The chapter ends by proposing *abstract garbage collection* as a technique to increase the precision of address-based observability by reducing the store to only contain reachable addresses in a certain state.

Chapter 4: Freshness Analysis

A resource is *fresh* in a particular context if it was created in that context. Effects on fresh resources can never be observed outside that context. We start this chapter with the observation that address-based observability from the previous chapter can be restated in terms of *address freshness*. Basing observability on address freshness is not always ideal because in a typical static analysis setting termination is primarily guaranteed by allowing resources to be allocated at addresses that are already in use, and therefore not fresh. To improve the precision of side-effect analysis, this chapter introduces two additional *scope-based* characterizations of freshness. *Variable freshness* is based on locality, and *object freshness* is an analysis that keeps track of the flow of objects between scopes and through object references. Their formalization forms the second main contribution of this work. We conclude the chapter by showing how to extend procedure side-effect analysis from the previous chapter with variable and object freshness.

Chapter 5: Escape Analysis

Variable and object freshness both rely on variables, which restricts freshness analysis of the previous chapter to the most recent application context. The reason is that procedure side-effect analysis uses stack reachability, and the stack embodies dynamic scope, while variables are lexically scoped. To allow scope-based freshness of variables and objects in underlying application contexts, this chapter formalizes an *escape analysis*. Escape analysis detects procedures that potentially escape their defining lexical scope. As long as no applied procedure escapes when traversing the stack, we are guaranteed that variables identify the same memory locations in the encountered stack frames. The chapter ends by showing how to incorporate the escape analysis into procedure side-effect analysis.

Chapter 6: Purity Analysis

This chapter presents our third main contribution, *purity analysis*, as an application of procedure side-effect analysis. The goal of purity analysis is to summarize the *side-effecting behavior of procedures* with respect to the rest of the program. The analysis identifies procedures that *generate* observable write effects, and procedures that *depend* on external side effects. Based on their side-effecting behavior, we *classify* procedures as either **pure**, **observer**, or **procedure**. A procedure is **pure** if none of its applications generate or depend upon externally observable side effects. A procedure is an **observer** as soon as one of its applications depends on an external side effect, but none of its applications generate observable side effects. Otherwise, a procedure is classified as **procedure**.

Chapter 7: Implementation

In this chapter we present a broad overview of our proof-of-concept implementation of the analyses of the previous chapters. We discuss the design and architecture of the resulting tool, and the techniques required to improve performance over a naive implementation. We zoom in on implementation details that are important to understand how different analysis configurations are realized.

Chapter 8: Evaluation

We applied our proof-of-concept implementation to a set of benchmarks. In this chapter we discuss the outcome of these experiments in terms of *correctness*, *soundness*, *precision*, and *performance*. We establish two

configurations for experimentation: a *concrete configuration* in which fully precise answers are obtained, and an *abstract configuration* that is guaranteed to be finite, but at the cost of introducing imprecision. We also give an overview of the benchmarks used for testing and evaluation.

Chapter 9: Purity Analysis For JavaScript

This chapter explains how our approach for purity analysis can be transposed to JavaScript. Although Scheme and JavaScript share some features such as higher-order procedures ("functions"), JavaScript adds features such as dynamic property expressions and prototypal inheritance. Although we follow the same general outline as for Scheme, we discuss the differences in design and implementation. Using a proof-of-concept implementation, we not only experiment on several JavaScript benchmarks, but also on some benchmarks we found in related work for comparison.

Chapter 10: Conclusion

We conclude this dissertation by restating the contributions and limitations of this work, guided by the outcome of our experiments. We discuss technical and applicative avenues of future work. On the technical side, we identify shortcomings of the current approach and possible improvements for the analyses and techniques we introduced. In terms of applications, we envision that analyses and techniques for parallelization, memoization, and referential transparency would fit well in our modular framework as client analyses.

1.7 Supporting Publications

There are two publications that support this dissertation directly. We discuss them briefly and highlight their relevance to this dissertation.

Detecting Function Purity in JavaScript [Nicolay et al., 2015]

This paper was presented at the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM2015) in Bremen, Germany, and was awarded the "Best Paper Award" at that same conference. It introduces the central idea to base a side-effect analysis and related analyses on an effect-tracking pushdown flow analysis that enables reasoning over

the stack to discover how and to which extent effects are linked to procedures. The main ideas of this paper appear in Chapter 3, but using Scheme instead of JavaScript. It also introduces freshness analysis for variables and objects, which is the subject of Chapter 4. The goal of [Nicolay et al., 2015] is to detect function purity in JavaScript, where function purity is defined as the absence of observable write effects. Chapter 6 in this dissertation reuses some of these ideas, but widens the scope considerably by also dealing with dependence on external side effects. The results reported on in this work are also more fine-grained, given our use of a ternary rather than a binary classification scheme for purity.

Automatic Parallelization of Side-Effecting Higher-Order Scheme Programs [Nicolay et al., 2011]

This paper was presented at the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM2011) in Williamsburg, VA, USA. It describes a program transformation that automatically parallelizes higher-order, side-effecting Scheme programs. The transformation has to be instantiated with an interprocedural dependence analysis that exposes parallelization opportunities in a sequential program. The dependence analysis is based on the observation that whenever a read or write effect occurs, every procedure on the call stack has a dependence on that resource. This insight plays a crucial role in the side-effect analysis of Chapter 3. The paper builds on a static analysis technique (AAM) that is less precise than the techniques employed in this dissertation (AAC) when it comes to call/return precision. In Chapter 10 we conjecture that the parallelization technique from the paper can be implemented in terms of the framework of this dissertation without much difficulty.

Other Publications

There are five additional publications we mention [Marr et al., 2012, Nicolay et al., 2013, Stievenart et al., 2015, Van Es et al., 2016, Vandercammen et al., 2015] that document our academic track record with respect to topics that are treated in this dissertation. These topics include abstract machines, interpreters, and static analysis in the context of higher-order, imperative languages.

1.8 Technical Contributions

Two technical artifacts support the techniques and experimental results in this dissertation.

`purity.rkt`

`purity.rkt` is the main artifact specifically developed for this dissertation. It is a Racket implementation that contains all analyses presented in this work. Its ANF Scheme input language and semantics are a superset of SCHEME_0, which we present in Chapter 2. Additionally, it also contains the benchmark programs, test suites for mechanical verification, and code for the experiments as described in Chapter 8.

We provide more details about this implementation in Chapter 7. The artifact is publicly available at `https://github.com/jensnicolay/purity/tree/phd`.

`protopurity.js`

A second artifact supporting this dissertation is `protopurity.js`. This artifact is a module of JIPDA, a static analysis framework for JavaScript, written in JavaScript.

`protopurity.js` implements the same ideas and techniques we cover in this dissertation, but for a challenging subset of JavaScript instead of Scheme.

It was used as the supporting implementation for published work [Nicolay et al., 2015], and for performing the evaluation in Chapter 9 of this dissertation.

The artifact is publicly available at `https://github.com/jensnicolay/jipda/tree/phd/protopurity`.

Chapter 2

Syntax and Semantics of SCHEME$_0$

2.1 Introduction

This chapter presents a formal overview of the syntax and semantics of SCHEME$_0$, our core Scheme language which we use throughout most of this dissertation.

We start with an informal definition of the features of SCHEME$_0$, and explain how they relate to program evaluation (Section 2.2). We then specify the syntax of SCHEME$_0$ (Section 2.3). To simplify the presentation, the syntax is in administrative normal form, and the side-effecting primitive procedures we are interested in are special forms. The semantics of SCHEME$_0$ are expressed as an abstract machine that explores reachable states (Section 2.4). The machine is parameterized with heap and stack allocators so that it can be configured to express concrete and abstract semantics. The result of state exploration is a flow graph, which is a representation of the runtime behavior of a program (Section 2.5). Before concluding the chapter, we present a fully worked-out example of program evaluation and stack traversal.

2.2 Informal Overview of SCHEME$_0$

The majority of this dissertation uses SCHEME$_0$ as the language of discourse, with only Chapter 9 devoted to JavaScript. SCHEME$_0$ was designed to be a close variant of R5RS Scheme [Abelson et al., 1998]. It features higher-order, first-class procedures, compound data structures, and side effects. Despite this rich set of features, SCHEME$_0$remains a small, consistent, and elegant language.

SCHEME$_0$is a *higher-order* language, meaning that all values, including procedures, can be passed as an argument and returned as a result. It is also an *imperative* language that models a heap or *store* in which values can be accessed and modified. A store maps *locations* (often represented as *addresses*) to values, and expressions in an imperative language are allowed to change the store.

Side effects are all effects that occur when an expression is evaluated besides producing the resulting value itself. While non-termination and out-of-memory errors are considered side effects as well, we limit effects to the reading and writing of store locations.

Procedures are treated as *first-class* values in SCHEME$_0$, meaning that procedures can be stored in and retrieved from the store.

SCHEME$_0$ has *pairs* as the basic building block for creating compound data structures. A pair corresponds to an object with two fields named `car` and `cdr`. Although not always the most convenient solution in practice, theoretically speaking a pair is the simplest possible component from which to construct a data structure, and it is possible to express any other structure in terms of pairs [Strachey, 2000].

Name resolution is defined in terms of the static structure of the program, making SCHEME$_0$ a *lexically scoped* language. Certain constructs such as procedures (`lambda` expressions) introduce new scopes or binding contexts that are nested into their surrounding (parent) scope. All named declarations are added to the innermost ("current") scope. Name lookup also starts at the innermost scope and proceeds outward until the name is found.

At runtime, an *environment* allows names to be substituted with their value, indirectly, by mapping identifiers to locations in the store. Manipulation of environments honors lexical scoping by extending and restoring them at scope entry and exit. When evaluation steps into a scope, the environment is extended to allow binding of formal parameters and local variables in the extended environment, without touching previous bindings. When leaving a scope, the previous environment is restored.

Because procedures in SCHEME$_0$ are first-class values that can freely flow through a program, care must be taken that a procedure's free variables remain lexically scoped. Free variables of a procedure are variables that are not local to a procedure, but that are declared in an outer lexical scope. To ensure that the static lookup chain is respected, a procedure evaluates to a *closure*, which is the procedure itself coupled to the environment in which it is defined. When the procedure is applied, it is this environment which is extended instead of the environment of the caller.

In the context of SCHEME$_0$, we use the term *procedure* to signify a syntactic occurrence of the `lambda` special form in a program, and prefer the use of the term *closure* to designate a `lambda` expression coupled to its static environment.

Finally, SCHEME$_0$ is an *applicative-order* language, so upon procedure application the arguments are evaluated and bound to parameters before entering the body of the procedure.

One notable omission in the semantics of SCHEME$_0$ is support for first-class control operators, of which Scheme's `call-with-current-continuation` is the exemplar. We revisit this limitation in Section 2.6, where we provide more technical details of why this limitation exists and why it is not so severe in practice.

Although we focus on SCHEME$_0$ in this dissertation, the ideas and techniques we present are applicable in other higher-order, imperative languages as well. In Chapter 9 we transpose our work to JavaScript.

2.3 Syntax

We work on SCHEME$_0$, a core Scheme language with first-class procedures, let binding, mutable variables, and mutable pairs. Although SCHEME$_0$ is a small language, it contains the essential features of a higher-order imperative programming language. The syntax of SCHEME$_0$ is depicted in Figure 2.1.

To simplify the presentation, SCHEME$_0$ is in Administrative Normal Form (ANF). Also for ease of exposition, we turn some constructs that are usually implemented as primitive procedures into syntax. We briefly discuss these qualities in the next two sections.

2.3.1 Administrative Normal Form

SCHEME$_0$ (Figure 2.1) is in Administrative Normal Form (ANF) [Flanagan et al., 1993]. ANF restricts a language by restricting most expression positions to atomic expressions. Only `let`-bound expressions and expressions in tail position are allowed to be non-atomic, or compound, expressions. This restriction is only cosmetic, because all programs in direct style can be expressed in ANF.

Atomic expressions help simplify the operation of the abstract machine. As we explain in Section 2.4, atomic expressions can be immediately evaluated. Unlike compound expressions, an interpreter does not need to step into

$$
\begin{array}{rll}
e \in \mathsf{Exp} ::= & æ & \text{[atomic expression]} \\
| & (f\ æ) & \text{[application]} \\
| & (\texttt{let}\ ((v\ e_0))\ e_1) & \text{[binding]} \\
| & (\texttt{set!}\ v\ æ) & \text{[set variable]} \\
| & (\texttt{cons}\ æ_1\ æ_2) & \text{[pair]} \\
| & (\texttt{car}\ v) \mid (\texttt{cdr}\ v) & \text{[get field]} \\
| & (\texttt{set-car!}\ v\ æ) \mid (\texttt{set-cdr!}\ v\ æ) & \text{[set field]} \\
f, æ \in \mathsf{Atom} ::= & v & \text{[variable]} \\
| & lam & \text{[lambda expression]} \\
lam \in \mathsf{Lam} ::= & (\lambda (v)\, e) & \\
v \in \mathsf{Var} = & \text{a set of identifiers} &
\end{array}
$$

Figure 2.1: Syntax of input language SCHEME$_0$.

atomic expressions, and "remember" where to continue after evaluating a subexpression. For function applications, for example, ANF ensures that all operands are atomic expressions, greatly simplifying the semantics. The same advantage of atomic expressions simplifies the semantics of `set!`, `set-car!`, etc.

A transformation to ANF breaks complex expressions down into a sequence of simpler intermediate subexpressions by introducing unary `let` expressions (i.e., containing one binding). It also imposes an order of evaluation. Due to the simplification that ANF brings, the abstract machine we define in Section 2.4 only has to remember something when it has to evaluate a `let` expression. The machine has to remember that after evaluating the value to be bound, it has to continue with the body of the `let`. Example 2.1 shows an input program in direct style, and the resulting program after conversion into ANF.

Example 2.1. An expression for calculating the n-th Fibonacci number is shown in Figure 2.2a. The normalized result is depicted in Figure 2.2b. In this example, variables with prefix `_p` are unique variables introduced by the transformation. In a similar manner ANF conversion also breaks down other types of conditional expressions, declarations, mutations and procedure bodies by explicitly naming subexpressions. □

```
(if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))
```

(a) Input program in direct style.

```
(let ((_p0 (< n 2)))
  (if _p0
      n
      (let ((_p1 (- n 1)))
        (let ((_p2 (fib _p1)))
          (let ((_p3 (- n 2)))
            (let ((_p4 (fib _p3)))
              (+ _p2 _p4)))))))
```

(b) Program after conversion to ANF.

Figure 2.2: Example of ANF conversion.

2.3.2 Side-effecting Special Forms

In SCHEME$_0$, all side-effecting constructs are implemented as special forms. Scheme implementations already implement the `set!` construct for variable mutation as a special form. In our syntax, the primitives operating on pairs (`cons`, `car`, `set-car!`, ...) are special forms too, instead of first-class primitive procedures. This enables our abstract machine specification to treat these special forms as syntax, i.e., at the same level as other expressions. Otherwise, a distinct category of primitives would be required, further complicating procedure application, which usually is already one of the most complicated semantic operations in an abstract machine.

Treating primitives as syntax renders SCHEME$_0$ less expressive, but in practice this is not a problem, as the following example makes clear.

Example 2.2. Suppose we are confronted with the following fragment of code, in which primitive `car`, passed as an argument to procedure `map`, is used as a first-class value.

```
(map car lst)
```

This fragment can be transformed into the equivalent, but more verbose fragment shown below.

```
(map (lambda (x) (car x)) lst)
```

This transformation works for other primitives as well. □

2.4 Semantics

The small-step semantics of SCHEME$_0$ is defined as an abstract machine that transitions between states [Felleisen and Friedman, 1987]. As the abstract machine describes the individual steps required to evaluate programs expressed in SCHEME$_0$, this type of semantics is referred to as *small-step* operational semantics. The abstract machine is constructed in a manner that allows functional reasoning about imperative constructs in the language.

Starting from an initial state, which represents the program to be evaluated, the machine continually transforms machine states into possible successor states, generating a graph of all potentially reachable states (Section 2.5). Every state corresponds to a snapshot of the program state while the machine is evaluating the program. Examining the resulting graph therefore enables us to determine runtime properties of the program that the machine interpreted. If a property of interest is not immediately captured by regular program interpretation, then the semantics can be *instrumented* by maintaining the derived property throughout the abstract machine states. We do this in our side-effect analysis of Chapter 3, when we extend our abstract machine to register effects that occur during evaluation. An analysis can traverse the resulting graph of program states, collecting the results of interest.

In the remainder of this section we detail the operation of the abstract machine.

2.4.1 State Space

Figure 2.3 depicts the state space of the abstract machine. The machine is a variation on the CESIK$^\star\Xi$ machine described by Johnson and Van Horn [2014]. Our core flow analysis adds configurable stack allocation to their Abstracting Abstract Control (AAC) approach.

The abstract machine is in eval-continuation style (**ev**–**ko**). A machine state is either an evaluation state (**ev**), or a continuation state (**ko**). In an evaluation state, the machine evaluates an expression e of SCHEME$_0$ in environment ρ. In a continuation state, the machine is ready to continue evaluation with a value d it has just computed.

$$
\begin{array}{rll}
\varsigma \in \mathit{State} & ::= \mathbf{ev}(e, \rho, \sigma, \iota, \kappa, \Xi) & \text{[eval state]} \\
& \mid \quad \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi) & \text{[kont state]} \\
\rho \in \mathit{Env} & = \mathsf{Var} \rightharpoonup \mathit{Addr} & \text{[environment]} \\
\sigma \in \mathit{Store} & = \mathit{Addr} \to D & \text{[store]} \\
d \in D & = \mathcal{P}(\mathit{Clo} + \mathit{Addr} + \mathit{Pair}) & \text{[value]} \\
\mathit{clo} \in \mathit{Clo} & ::= \mathbf{clo}(\mathit{lam}, \rho) & \text{[closure]} \\
p \in \mathit{Pair} & ::= \mathbf{cons}(d_{\text{car}}, d_{\text{cdr}}) & \text{[pair]} \\
\iota \in \mathit{LKont} & = \mathit{Frame}^* & \text{[frame]} \\
\phi \in \mathit{Frame} & ::= \mathbf{let}(v, e, \rho) & \text{[binding frame]} \\
\kappa \in \mathit{Kont} & ::= \epsilon \mid \tau & \text{[application context]} \\
\Xi \in \mathit{KStore} & = \mathit{Ctx} \to \mathcal{P}(\mathit{LKont} \times \mathit{Kont}) & \text{[stack store]} \\
a \in \mathit{Addr} & = \text{a set of addresses} & \\
\tau \in \mathit{Ctx} & = \text{a set of stack addresses} &
\end{array}
$$

Figure 2.3: Abstract state-space of the flow analysis.

Environment and Store

An environment ρ maps variables to addresses. Addresses represent a location in the store σ, and the store maps addresses to values. Looking up the value of a variable therefore consists of first looking up that variable's address, and then referencing this address in the store.

Values

Values in our semantics are closures (clo), addresses (a), and pairs (p). A closure is a lambda expression coupled to an environment. An address is a pointer to a pair, so addresses in general serve as handles to objects. Closures and addresses are first-class values that can be manipulated directly in a program. A pair is a tuple of values, and is manipulated by the syntax operating on pairs. We only need to model the `car` and `cdr` fields for pairs in our language. In a more general setting, which for example supports vectors and objects, the set of field names can be considerably larger, or infinite.

Stacks

The stack is modeled as a local continuation (ι) delimited by a meta-continuation (κ). The local continuation is a (possibly empty) list of frames, and acts like a "regular" stack, on top of which frames can be pushed and from which topmost frames can be popped. The meta-continuation is either empty (ϵ), or a stack address (τ) pointing to underlying stacks stored in a stack store (Ξ). Traversing the stack from top to bottom happens by first traversing the local continuation ι (if not empty), and then looking up the underlying stacks by dereferencing the stack address τ in the stack store Ξ. This process repeats until the meta-continuation is not a stack address but the empty meta-continuation ϵ, which means the bottom of the stack is reached.

2.4.2 Abstract Semantics

In this dissertation, the information we want to extract from a program's flow graph is the following:

- Value flow: what are the possible values that expressions in a program may attain?
- Control flow: what are the possible paths through a program?
- Effects: what resources are read and written by the evaluation of expressions?
- Freshness: which resources are created during a particular evaluation context, and which resources existed beforehand?

To be able to extract this program information from a flow graph, the graph has to be finite. The problem with computing a flow graph using an abstract machine as described above, is that the graph can become unpractically large, or even infinite. We therefore need a technique to ensure termination of evaluation, while still obtaining useful and consistent results about the behavior of programs.

Abstract interpretation is a theory and a framework that offers a solution to the problem of costly or uncomputable runtime properties. It was pioneered as a formal method for the analysis of programs in Cousot and Cousot [1977]. Although the concrete semantics of a non-trivial program are not computable with finite resources, abstract interpretation nevertheless offers a framework for getting useful answers to non-trivial questions

about programs. The answers we obtain from abstract interpretation will necessarily be approximations due to the undecidability of the questions asked.

In the context of this dissertation we want static analysis to *conservatively approximate* control flow, value flow, effects, and freshness. Conservative means erring on the safe side in order to remain faithful to the original semantics. For example, if during procedure application a resource is accessed or modified, the analysis must reflect this possibility, but the analysis may be imprecise in the sense that it overestimates the set of resources read or written at a specific call site.

The state machine we presented is actually an *abstraction* (Appendix B.1) of an abstract state machine that is a concrete interpreter for SCHEME_0. The telltale sign of this is the fact that the range of the value store and stack store are powersets, which are lattices (Example B.6). This is necessary, because allocators are allowed to choose addresses that are already in use, leading to *weak updating* of values in these stores. Weak updating pigeonholes old and new values at the same address by joining them through join operator $\sqcup$, requiring the values to form a lattice (Appendix B.2). The store address allocator and stack address allocator are parameters of the machine, and can be used to vary the abstractions and resulting semantics.

2.4.3 Concrete Semantics

The semantics we present in this chapter are the abstract semantics, using weak updates through lattice operator $\sqcup$. However, the abstract machine simulates concrete semantics when configured with stack and store allocators that always allocate fresh addresses, and when *strong update* is used for both stores. Strong updating replaces the previous value with a new value. In this setting the powersets in the range of the stores are guaranteed to be singleton sets. Concrete semantics has full precision in that joins in control and value flow are avoided, almost always resulting in a flow graph that is a linear sequence of states.

2.4.4 Program Injection

The injection function $\mathcal{I} : \mathsf{Exp} \to \mathit{State}$ turns an expression into an initial evaluation state with empty environment, store, local continuation, meta-

continuation, and stack store.

$$\mathcal{I}(e) = \mathbf{ev}(e, [], [], \langle\rangle, \epsilon, [])$$

2.4.5 Atomic Evaluation

The atomic evaluation function $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \to D$ evaluates atomic expressions into a value. Atomic expressions (Section 2.3.1) are expressions that can be evaluated in a bounded number of steps without modifying the environment, store, or stack.

In the semantics of SCHEME$_0$, there are two kinds of atomic expressions: procedures and variables.

1. A procedure evaluates to closure, which is a tuple of the procedure and the environment.

$$\mathcal{A}(\mathit{lam}, \rho, \sigma) = \{\mathbf{clo}(\mathit{lam}, \rho)\}$$

2. Looking up the value of a variable happens by first looking up the address of the variable in the environment, and then looking up the value associated with that address in the store.

$$\begin{aligned}\mathcal{A}(v, \rho, \sigma) &= \sigma(a)\\ \text{where } a &= \rho(v)\end{aligned}$$

2.4.6 Address Allocation

The address allocation functions *allocVar* and *allocPair* choose store addresses for newly bound variables and pairs, respectively. The address allocation functions are opaque parameters in this semantics, and can be used to tune performance and precision of the abstract machine and resulting flow analysis. For maximum flexibility in choosing addresses, we pass almost the entire program state to the allocator functions.

$$\begin{aligned}\mathit{allocVar} &: \mathsf{Var} \times \mathsf{Exp} \times \mathit{Store} \times \mathit{LKont} \times \mathit{Kont} \times \mathit{KStore} \to \mathit{Addr}\\ \mathit{allocPair} &: \mathsf{Exp} \times \mathit{Store} \times \mathit{LKont} \times \mathit{Kont} \times \mathit{KStore} \to \mathit{Addr}\end{aligned}$$

Concrete semantics can be defined by letting the allocator return fresh addresses, for example by choosing addresses from the set of natural numbers

and returning a number that is not in use.

$$\begin{aligned} \mathit{Addr} &= \mathbb{N} \\ \mathit{allocVar}(v, e, \sigma, \iota, \kappa, \Xi) &= 1 + \max(\mathrm{Dom}(\sigma)) \\ \mathit{allocPair}(e, \sigma, \iota, \kappa, \Xi) &= 1 + \max(\mathrm{Dom}(\sigma)) \end{aligned}$$

For defining finite abstract semantics, the allocator can use the involved variables or expressions (i.e., syntax) directly as addresses. This allocation strategy is called 0CFA [Shivers, 1991].

$$\begin{aligned} \mathit{Addr} &= \mathsf{Exp} \\ \mathit{allocVar}(v, e, \sigma, \iota, \kappa, \Xi) &= v \\ \mathit{allocPair}(e, \sigma, \iota, \kappa, \Xi) &= e \end{aligned}$$

2.4.7 Stack Address Allocation

The stack address allocation function *allocCtx* is also a parameter of the language semantics, and returns stack addresses for allocating stacks in the stack store. Stack addresses are generated at call sites, and therefore represent *application contexts*. A *call stack* can be obtained by tracing out all reachable stack addresses in the stack store, starting from the *topmost context* directly contained in the state. Stack traversal terminates when the empty meta-continuation (or *root context*) ϵ is reached. A program starts evaluation and, if evaluation terminates normally, finishes evaluation in the root context. In SCHEME$_0$, the root context corresponds to the top-level part of a program that is not within any procedure. Example 2.3 on page 29 visualizes some of the terminology.

As with the store allocator, we pass a large part of the program state to the stack store allocator function for flexibility reasons.

$$\mathit{allocCtx} : \mathsf{Exp} \times \mathit{Clo} \times \mathit{D} \times \mathit{Store} \times \mathit{Env} \to \mathit{Addr}$$

We next discuss two stack address allocators, AAC and P4F, used in related works.

AAC Stack Allocation

The original Abstracting Abstract Control (AAC) approach [Johnson and Van Horn, 2014] uses an allocation function that returns stack addresses consisting of all components that influence the outcome of a procedure

application, i.e., the entire calling context (assuming the absence of first-class control). To avoid unnecessary merging in return flow, our approach adds an additional component corresponding to the syntactic call expression e. This more refined allocation strategy defines the concrete semantics of the language completely, while ensuring maximal call/return precision for its semantics, i.e., full precision modulo any abstraction.

$$\begin{aligned} \mathit{Ctx} &= (\mathsf{Exp} \times \mathit{Clo} \times D \times \mathit{Store}) \\ \mathit{allocCtx}(e, \mathit{clo}, d, \sigma, \rho) &= (e, \mathit{clo}, d, \sigma) \end{aligned}$$

P4F Stack Allocation

A more lightweight representation of the calling context has recently been proposed in the P4F[1] approach by Gilray et al. [2016]. P4F refines the AAC approach by using a different stack allocator that offers the same precision as the original AAC allocation strategy when both the value store and stack store are global, but only includes the applied procedure and its definition environment extended with parameter bindings. As a result, the worst-case computational cost of flow analysis drops from $O(n^8)$ for AAC to $O(n^3)$ for P4F when using 0CFA and global stores (Section 7.4.1).

$$\begin{aligned} \mathit{Ctx} &= (\mathsf{Lam} \times \mathit{Env}) \\ \mathit{allocCtx}(e, (\mathit{lam}, \rho), d, \sigma, \rho') &= (\mathit{lam}, \rho') \end{aligned}$$

2.4.8 Transition Relation

Using the auxiliary functions from previous sections, we can now define the transition relation $(\longmapsto) \subseteq \mathit{State} \times \mathit{State}$ of our abstract machine. Rules for transitions from evaluation states (**ev**) correspond with the different syntactic cases, while rules for transitions from continuation states (**ko**) correspond with the different kinds of continuations. We list the different cases for $\longmapsto$.

1. Evaluation of an atomic expression $æ$ continues with the value of the atomic expression computed by the atomic evaluator $\mathcal{A}$ (Section 2.4.5).

$$\mathbf{ev}(æ, \rho, \sigma, \iota, \kappa, \Xi) \longmapsto \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi) \quad \text{where } d = \mathcal{A}(æ, \rho, \sigma) \qquad [\text{E-ATOM}]$$

[1] P4F stands for "pushdown control-flow analysis for free"

2. To evaluate a variable binding, the machine moves to a state that will evaluate the value expression e_0, while pushing a frame to remember to continue with the evaluation of body expression e_1.

$$\mathbf{ev}(\llbracket\texttt{(let ((}v\ e_0\texttt{))}\ e_1\texttt{)}\rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \longmapsto \mathbf{ev}(e_0, \rho, \sigma, \mathbf{let}(v, e_1, \rho) : \iota, \kappa, \Xi) \quad \text{[E-LET]}$$

3. Evaluating a variable mutation requires atomically evaluating value expression $æ$ and assigning the obtained value d to a name. In our semantics, the result of variable mutation is the assigned value d.

$$\begin{aligned} \mathbf{ev}(\llbracket\texttt{(set!}\ v\ æ\texttt{)}\rrbracket, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto \mathbf{ko}(d, \sigma', \iota, \kappa, \Xi) \qquad \text{[E-SET]} \\ \text{where } d &= \mathcal{A}(æ, \rho, \sigma) \\ a &= \rho(v) \\ \sigma' &= \sigma \sqcup [a \mapsto d] \end{aligned}$$

4. Evaluating a function call involves atomically evaluating the operator f to a closure and the operand $æ$ to a value, and binding the operand value d_{arg} by extending the static environment of the closure ρ'' and the store. The current stack (ι, κ) is allocated in the stack store at freshly allocated stack address τ, and evaluation moves to the body of the lambda-term within the closure e_0 with an empty local continuation and application context τ.

$$\begin{aligned} \mathbf{ev}(\overbrace{\llbracket\texttt{(}f\ æ\texttt{)}\rrbracket}^{e}, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto \mathbf{ev}(e_0, \rho', \sigma', \langle\rangle, \tau, \Xi') \qquad \text{[E-APP]} \\ \text{where } clo &\in \mathcal{A}(f, \rho, \sigma) \\ d_{\text{arg}} &= \mathcal{A}(æ, \rho, \sigma) \\ \mathbf{clo}(\llbracket\texttt{(}\lambda\texttt{(}v\texttt{)}\, e_0\texttt{)}\rrbracket, \rho'') &= clo \\ a &= allocVar(v, e, \rho, \sigma, \iota, \kappa, \Xi) \\ \rho' &= \rho''[v \mapsto a] \\ \sigma' &= \sigma \sqcup [a \mapsto d_{\text{arg}}] \\ \tau &= allocCtx(e, clo, d_{\text{arg}}, \sigma, \rho') \\ \Xi' &= \Xi \sqcup [\tau \mapsto (\iota, \kappa)] \end{aligned}$$

5. Evaluating the allocation of a pair requires atomically evaluating the `car` and `cdr` value.

$$\mathbf{ev}(\overbrace{[\![\texttt{(cons } æ_1 \ æ_2\texttt{)}]\!]}^{e}, \rho, \sigma, \iota, \kappa, \Xi) \longmapsto \mathbf{ko}(\{a\}, \sigma', \iota, \kappa, \Xi) \quad \text{[E-CONS]}$$
$$\text{where } d_{\text{car}} = \mathcal{A}(æ_1, \rho, \sigma)$$
$$d_{\text{cdr}} = \mathcal{A}(æ_2, \rho, \sigma)$$
$$a = \mathit{allocPair}(e, \rho, \sigma, \iota, \kappa, \Xi)$$
$$\sigma' = \sigma \sqcup [a \mapsto \{\mathbf{cons}(d_{\text{car}}, d_{\text{cdr}})\}]$$

6. To take the `car` field of a pair, the abstract machine first atomically evaluates expression v to obtain a pointer d_{car} to a pair, and then returns the first field of the pair. (The rule for getting the `cdr` field is similar.)

$$\mathbf{ev}(\overbrace{[\![\texttt{(car } v\texttt{)}]\!]}^{e}, \rho, \sigma, \iota, \kappa, \Xi) \longmapsto \mathbf{ko}(d_{\text{car}}, \sigma, \iota, \kappa, \Xi) \quad \text{[E-CAR]}$$
$$\text{where } a \in \mathcal{A}(v, \rho, \sigma)$$
$$\mathbf{cons}(d_{\text{car}}, \_) \in \sigma(a)$$

7. To evaluate a mutation of the `car` field of a pair, the abstract machine atomically evaluates the pair d and value expression $æ$, and replaces the first field of the obtained pairs with the computed value. (The rule for setting the `cdr` field is similar.)

$$\mathbf{ev}(\overbrace{[\![\texttt{(set-car! } v \ æ\texttt{)}]\!]}^{e}, \rho, \sigma, \iota, \kappa, \Xi) \longmapsto \mathbf{ko}(d_{\text{car}}, \sigma', \iota, \kappa, \Xi) \quad \text{[E-SET-CAR]}$$
$$\text{where } a \in \mathcal{A}(v, \rho, \sigma)$$
$$d_{\text{car}} = \mathcal{A}(æ, \rho, \sigma)$$
$$\mathbf{cons}(\_, d_{\text{cdr}}) \in \sigma(a)$$
$$\sigma' = \sigma \sqcup [a \mapsto \{\mathbf{cons}(d_{\text{car}}, d_{\text{cdr}})\}]$$

8. When the machine has to continue with a value d and the local continuation is empty, the machine dereferences the stack address κ to obtain an underlying stack (ι', κ'). If no stacks are found in the stack store, then the machine has reached a program exit and halts, and d is the result value of the program. In all other cases the machine

has reached a function exit, and will keep popping the stack until a non-empty local continuation is available.

$$\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) \longmapsto \mathbf{ko}(d, \sigma, \iota', \kappa', \Xi) \quad \text{where } (\iota', \kappa') \in \Xi(\kappa) \qquad \text{[K-RET]}$$

9. With a `let`-created continuation as the topmost frame, execution returns to the body of the `let` that created it, with an extended environment and store.

$$\mathbf{ko}(d, \sigma, \mathbf{let}(v, e, \rho) : \iota, \kappa, \Xi) \longmapsto \mathbf{ev}(e, \rho', \sigma', \iota, \kappa, \Xi) \qquad \text{[K-LET]}$$
$$\text{where } a = allocVar(v, e, \rho, \sigma, \iota, \kappa, \Xi)$$
$$\rho' = \rho[v \mapsto a]$$
$$\sigma' = \sigma \sqcup [a \mapsto d]$$

2.5 Flow Graph Construction

Program properties can be determined by reasoning about all reachable machine states. We therefore construct a *flow graph* representing program evaluation, in which nodes are reachable states, and edges are transitions between states. Evaluation of an expression e can be expressed as computing the reflexive transitive closure of $\longmapsto$ after injection (Section 2.4.4).

$$\mathcal{E}(e) = \{\varsigma \mid \mathcal{I}(e) \longmapsto^* \varsigma\}$$

The definition of flow graph G_e for expression e then is as follows:

$$(\varsigma \to \varsigma') \in G_e \iff \varsigma \in \mathcal{E}(e) \text{ and } \varsigma \longmapsto \varsigma'$$

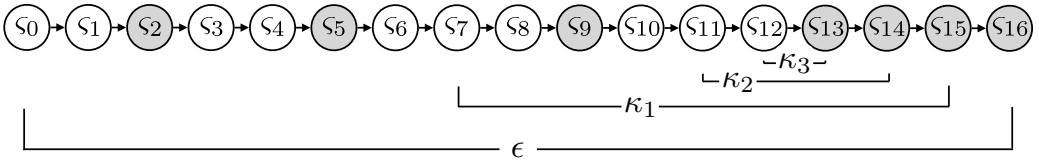
Static analysis requires a finite flow graph for the program under analysis. Finiteness is guaranteed by plugging in finite sets for Var, $Addr$, and Ctx into the state space of the analysis (Figure 2.3). The entire state space then becomes finite as well, and the function computing the reflexive transitive closure of $\longmapsto$ is monotonic and therefore has a least fixpoint.

2.5.1 Evaluation Example

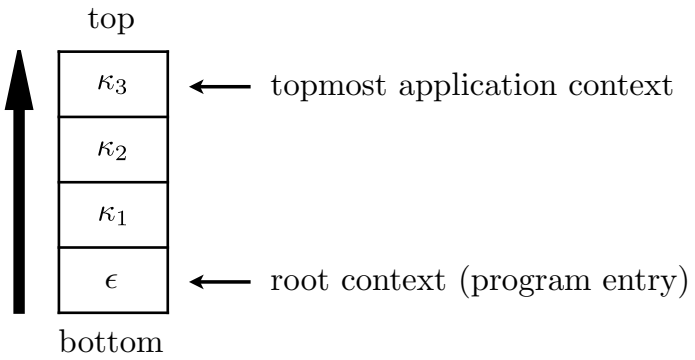
Before concluding the chapter, we give an extensive example of program evaluation using the abstract machine we presented. We also show how stack traversal is performed.

```
1 (let ((h (lambda (y)
2              y)))
3   (let ((f (lambda ()
4              (let ((g (lambda (x)
5                         (x x))))
6                (g h)))))
7     (f)))
```

(a) Example program in which procedure `f` calls procedure `g`, and procedure `g` calls procedure `h`.



(b) Flow graph of the program, with dynamic extents of application contexts drawn below (details of states and contexts in the running text). Unshaded states are evaluation states (**ev**), and shaded states are continuation states (**ko**).



(c) Call stack in state ς_{12}, when evaluating expression `y` on line 2.

Figure 2.4: Example program and depiction of flow graph and call stack for Example 2.3.

Example 2.3. Consider the program depicted in Figure 2.4a. Assume the abstract machine from Section 2.4 has been configured to express abstract semantics, with 0CFA as allocator for store values, and P4F as allocator for stacks. 0CFA uses AST nodes as addresses (variables or `cons` expressions). P4F allocates a stack using the applied procedure and the extended environment.

Injecting this program into the machine results in an initial state ς_0. The machine then explores all reachable states starting from the initial state, leading to the graph depicted in Figure 2.4b. The detailed composition of the states, together with the transitions between them, is given at the end of this example.

Figure 2.4c shows the call stack in state ς_{12}. The call stack in a state can be obtained by only considering the reachable meta-continuations, ignoring local continuations. This indeed yields the call stack because continuations are delimited at procedure calls, and meta-continuations are represented as stack addresses allocated at call sites (transition rule [E-APP]). A stack address therefore corresponds to an application context.

The topmost application context κ_3 is the meta-continuation of ς_{12}. We can move "down" the stack by dereferencing κ_3 in the stack store, which yields underlying stack $(\langle\rangle, \kappa_2)$, i.e., a pair of the empty local continuation $\langle\rangle$ and meta-continuation κ_2. Repeating this process of dereferencing leads to the bottom of the stack $(\langle\rangle, \epsilon)$. Because P4F adds the applied procedure to the application context, it is straightforward to determine that expression `y` in ς_{12} is evaluated in the dynamic extent of procedures `h`, `g`, and `f`.

We end the example by giving the exploration of states worked out in full detail.

0. $\mathbf{ev}(\llbracket\texttt{(let ((h (lambda (y) y))) (let ((f ...)))))}\rrbracket, \rho_0, \sigma_0, \langle\rangle, \epsilon, \Xi_0)$

$$\text{where } \rho_0 = [] \quad \sigma_0 = [] \quad \Xi_0 = []$$

$$\downarrow \quad \text{[E-LET]}$$

1. $\mathbf{ev}(\overbrace{\llbracket\texttt{(lambda (y) y)}\rrbracket}^{\lambda_2}, \rho_0, \sigma_0, \phi_2 : \langle\rangle, \epsilon, \Xi_0)$

$$\text{where } \phi_2 = \mathbf{let}(\texttt{h}, \llbracket\texttt{(let ((f (lambda () (let ((g ...)))))))}\rrbracket, \rho_0)$$

$$\downarrow \quad \text{[E-ATOM]}$$

2. $\mathbf{ko}(clo_2, \sigma_0, \phi_2 : \langle\rangle, \epsilon, \Xi_0)$

$$\text{where } clo_2 = (\lambda_2, \rho_0)$$

$$\downarrow \quad [\text{K-LET}]$$

3. $\mathbf{ev}(\llbracket\texttt{(let ((f (lambda () (let ((g...)))))))}\rrbracket, \rho_1, \sigma_1, \langle\rangle, \epsilon, \Xi_0)$

$$\text{where } \rho_1 = \rho_0[\mathsf{h} \mapsto \mathsf{h}]$$
$$\sigma_1 = \sigma_1[\mathsf{h} \mapsto \{clo_2\}]$$

$$\downarrow \quad [\text{E-LET}]$$

4. $\mathbf{ev}(\overbrace{\llbracket\texttt{(lambda () (let ((g...))))}\rrbracket}^{\lambda_7}, \rho_1, \sigma_1, \phi_3 : \langle\rangle, \epsilon, \Xi_0)$

$$\text{where } \phi_3 = \mathbf{let}(\mathsf{f}, \llbracket\texttt{(f)}\rrbracket, \rho_1)$$

$$\downarrow \quad [\text{E-ATOM}]$$

5. $\mathbf{ko}(clo_7, \sigma_1, \phi_3 : \langle\rangle, \epsilon, \Xi_0)$

$$\text{where } clo_7 = (\lambda_7, \rho_1)$$

$$\downarrow \quad [\text{K-LET}]$$

6. $\mathbf{ev}(\llbracket\texttt{(f)}\rrbracket, \rho_2, \sigma_2, \langle\rangle, \epsilon, \Xi_0)$

$$\text{where } \rho_2 = \rho_1[\mathsf{f} \mapsto \mathsf{f}]$$
$$\sigma_2 = \sigma_1[\mathsf{f} \mapsto \{clo_7\}]$$

$$\downarrow \quad [\text{E-APP}]$$

7. $\mathbf{ev}(\llbracket\texttt{(let ((g (lambda (x) (x x)))) (g h))}\rrbracket, \rho_1, \sigma_2, \langle\rangle, \kappa_1, \Xi_1)$

$$\text{where } \kappa_1 = (\lambda_7, \rho_1)$$
$$\Xi_1 = [\kappa_1 \mapsto \{(\langle\rangle, \epsilon)\}]$$

$$\downarrow \quad [\text{E-LET}]$$

8. $\mathbf{ev}(\overbrace{\llbracket\texttt{(lambda (x) (x x))}\rrbracket}^{\lambda_{10}}, \rho_1, \sigma_2, \phi_4 : \langle\rangle, \kappa_1, \Xi_1)$

$$\text{where } \phi_4 = \mathbf{let}(\mathsf{g}, \llbracket\texttt{(g h)}\rrbracket, \rho_1)$$

$$\downarrow \quad [\text{E-ATOM}]$$

9. $\mathbf{ko}(clo_{10}, \sigma_2, \phi_4 : \langle\rangle, \kappa_1, \Xi_1)$

$$\text{where } clo_{10} = (\lambda_{10}, \rho_1)$$

$$\downarrow \quad [\text{K-LET}]$$

10. $\mathbf{ev}([\![\texttt{(g h)}]\!], \rho_3, \sigma_3, \langle\rangle, \kappa_1, \Xi_1)$

$$\begin{aligned}\text{where } \rho_3 &= \rho_1[\mathsf{g} \mapsto \mathsf{g}] \\ \sigma_3 &= \sigma_2[\mathsf{g} \mapsto \{clo_{10}\}]\end{aligned}$$

$$\downarrow \quad [\text{E-APP}]$$

11. $\mathbf{ev}([\![\texttt{(x x)}]\!], \rho_4, \sigma_4, \langle\rangle, \kappa_2, \Xi_2)$

$$\begin{aligned}\text{where } \rho_4 &= \rho_1[\mathsf{x} \mapsto \mathsf{x}] \\ \sigma_4 &= \sigma_3[\mathsf{x} \mapsto \{clo_2\}] \\ \kappa_2 &= (\lambda_{10}, \rho_4)\end{aligned}$$

$$\downarrow \quad [\text{E-APP}]$$

12. $\mathbf{ev}([\![\mathsf{y}]\!], \rho_5, \sigma_5, \langle\rangle, \kappa_3, \Xi_3)$

$$\begin{aligned}\text{where } \rho_5 &= \rho_0[\mathsf{y} \mapsto \mathsf{y}] \\ \sigma_5 &= \sigma_4[\mathsf{y} \mapsto \{clo_2\}] \\ \kappa_3 &= (\lambda_2, \rho_5) \\ \Xi_3 &= \Xi_2[\kappa_3 \mapsto \{(\langle\rangle, \kappa_2)\}]\end{aligned}$$

$$\downarrow \quad [\text{E-ATOM}]$$

13. $\mathbf{ko}(\{clo_2\}, \sigma_5, \langle\rangle, \kappa_3, \Xi_3)$

$$\downarrow \quad [\text{K-RET}]$$

14. $\mathbf{ko}(\{clo_2\}, \sigma_5, \langle\rangle, \kappa_2, \Xi_3)$

$$\downarrow \quad [\text{K-RET}]$$

15. $\mathbf{ko}(\{clo_2\}, \sigma_5, \langle\rangle, \kappa_1, \Xi_3)$

$$\downarrow \quad [\text{K-RET}]$$

16. $\mathbf{ko}(\{clo_2\}, \sigma_5, \langle\rangle, \epsilon, \Xi_3)$

□

2.6 Related Work

Felleisen and Friedman [1987] introduced the CESK abstract machine to express the semantics of higher-order, functional programs with side effects.

Shivers [1988] developed the first control-flow analysis for higher-order languages, k-CFA, using abstract interpretation. He developed k-CFA in the context of Scheme programs expressed in continuation-passing style. 0CFA is at the bottom of the k-CFA hierarchy, and is the simplest abstract allocation policy to implement. We describe and employ 0CFA as an abstract store allocator throughout this dissertation,

Might and Shivers [2006b] reformulated Shiver's denotational semantics into a small-step operational semantics, expressed as an abstract machine, and added garbage collection as an optimization.

Van Horn and Might [2010] devised the "Abstracting Abstract Machines" (AAM) approach to systematically construct static analyses based on abstract machine semantics.

Vardoulakis and Shivers [2010] formulated the first precise flow analysis for higher-order languages based on a pushdown model. Unlike a finite-state analysis, which typically only keeps track of a small number of called procedures, a pushdown analysis models call/return precisely.

At around the same time Earl et al. [2010] developed an alternative pushdown flow analysis (PDCFA) featuring a more explicit pushdown system at its core, which was extended afterwards to enable stack inspection (IPDCFA) [Earl et al., 2012].

Johnson and Van Horn [2014] revised the AAM approach into the "Abstracting Abstract Control" (AAC) approach, in order to tackle first-class control in the setting of pushdown flow analysis. AAC is based on abstract machines but does not require pushdown automata theory, and generalizes over both CFA2 and IPDCFA.

SCHEME$_0$ does not have support for control operations that actually require saving and restoring continuations in the store (e.g., Scheme's `call-with-current-continuation`). Control operations that respect a "proper" stack discipline (i.e., which can be expressed as one or more pop operations) do not present a problem in our approach. Statements and constructs like `return`, `try`–`catch`, exit continuations in general (Racket's `call-with-escape-continuation`), and basically all control-flow operations that can be implemented using some form of continuation marks [Clements and Felleisen, 2004] all fall within this category.

In the semantics we presented, saving continuations in the store is problematic because continuations can contain elements of the store, leading to structural recursion and non-terminating analyses. Johnson and Van Horn [2014] solves this problem by (re-)applying AAM (by indirecting through the store) to those continuations that need to be saved in the store.

Gilray et al. [2016] refines the AAC approach by choosing the smallest possible stack addresses that still result in maximal stack precision in the presence of global stores.

2.7 Conclusion

In this chapter we specified SCHEME$_0$ and its semantics. We expressed the semantics as an abstract state machine that transitions between states. These transitions are captured in a flow graph, which then can be examined to compute program properties of interest.

In order to simplify the development and formalization of the techniques described in the remainder of this dissertation, we presented SCHEME$_0$, a functional Scheme language with assignment and pairs. Although it is a small language, it contains the essential features of higher-order imperative programming languages, and therefore is sufficiently challenging for performing static analysis. Prototype implementations of ideas and techniques developed in the upcoming chapters (Chapter 7) support additional syntax, built-in data types such as vectors, and built-in primitives.

In Chapter 3 we extend the base semantics of this chapter by instrumenting the interpreter with effects.

Chapter 3

Side-Effect Analysis

> Happy is he who from effects
> can ascend to their causes.
>
> — Virgil

3.1 Introduction

In functional programming languages, an expression only has a value. In a language with imperative constructs, expressions not only compute values but are also allowed to affect the store. *Effects* describe how expressions affect the store, providing intensional information about what takes place during evaluation of the program [Nielson et al., 1999, Talpin and Jouvelot, 1992]. Everything else an expression does besides computing a value, is called a *side effect* of that expression.

While side effects may be straightforward to determine for most individual expressions in isolation, the interesting but difficult case is when procedure calls are involved.

- Procedures only generate effects upon application. Therefore our notion of procedure side effects — a procedure as a syntactic entity appearing in a program — is linked to the behavior of all of its applications at runtime.

- Procedures can call other procedures, and when an effect occurs an effect analysis necessarily need to traverse the stack to find all "active" procedures, i.e., procedures that are in the process of computing a

result. This is what Might and Prabhu [2009] formulated as Harrison's dependence principle [Harrison, 1989], which, reformulated in the context of side-effect analysis, states that a side effect occurs for all procedures on the call stack.

- Procedure applications establish contexts in a program that enable distinguishing between effects that are local to an application context and therefore not observable to the outside world, and all other effects that are observable outside that context. A procedure side effect is *observable* when the effect is on a resource that exists in the store of a caller. When side effects cannot be observed by callers of a procedure, they are *unobservable* for those callers. When it can be determined that a side effect is unobservable in a certain context, we say that the effect can be *masked*.

- Procedures are units of code that are executed frequently during program execution, and they can be named, exported, and memoized.

For these reasons, procedures are at a suitable level of code granularity to determine side effects, and therefore we primarily treat side effects at the level of procedures and procedure calls.

In this chapter we present an approach to detect procedure side effects in higher-order, imperative programs. We consider variables, objects, and object fields as *resources*. Variables and objects are stored at specific memory locations, and reading from and writing to memory locations are considered to be *effects*. To increase the effectiveness of side-effect analysis, the goal is to soundly mask as many effects as possible. This implies that observability of a side effect must be examined for every procedure application on the call stack. Therefore, our approach can, depending on the context, consider procedures that mutate memory locations or depend on mutable memory locations to be free from side effects.

We develop a formal definition of resources, effects, and observability through stack reachability in Sections 3.2 to 3.4.

3.1.1 Challenges

Procedure side-effect analysis involves determining for every effect whether it is observable by any caller on the stack at the point the effect occurs. Computing observability of side effects at compile-time is challenging, and

particularly so in the presence of objects and higher-order procedures that may flow freely through a program, as the following examples illustrate.

Example 3.1. The following program defines procedure `f`, which has two nested procedures `g` and `h`.

```
1 (define (f)          ; no observable write effect
2   (define (g p)      ; generates observable write effect
3     (h p))
4   (define (h q)      ; generates observable write effect
5     (set-car! q 3))
6   (let ((o (cons 1 2)))
7     (g o)))
8 (f)
```

Procedure `f` creates a pair `o` (line 6) and passes this object to procedure `g` (line 7), which in turn passes it to procedure `h` (line 3). Function `h` then modifies the `car` field of the pair on line 5, which constitutes an observable side effect, as writes to this field are observable to the caller `g`. This means that the mutated pair exists in the caller store that is in effect when the application at line 3 is evaluated.

When the write to `q` occurs, not only procedure `h` is active, but procedures `g` and `f` are also on the call stack. For the same reason as before, the caller of `g` (i.e., procedure `f`) also observes this effect, so the field write is also an observable side effect of procedure `g`. However, the effect is not observable by the callers of `f`, as `f` created the pair and holds the only reference to it. This example demonstrates that determining whether an effect is observable in the context of a procedure application requires traversing the full call stack that is in place when the effect occurs. □

The propagation of an effect along the call stack "stops" at the point where the effect is or becomes unobservable. In Example 3.1, no caller of `f` will ever observe the write effect generated by `set-car!`, so all procedures on the call stack "below" a particular application of `f` are not impacted by this effect.

Example 3.2. The following program features a higher-order procedure `f` that applies its parameter `h`.

```
1 (define (f h)        ; generates observable write effect
2   (h))
3 (let ((z #t))
```

```
4  (define (g)       ; generates observable write effect
5    (set! z #f))
6  (f g))
```

Procedure g mutates its free variable z, which results in a side effect that is observable by callers of g. Procedure f receives a closure of procedure g as argument (line 6) and applies it on line 2. Because variable z existed in the caller store at the time that f was applied on line 6, the write effect of g on z is also observable by the caller of f, although f itself cannot directly access z. This illustrates that our side-effect analysis needs to be able to handle first-class procedures, closures, and lexical scopes. □

The previous example shows that our definition of observability does not depend on accessibility of the resource *in the program*. Instead, observability is defined on the level of the entire program state. We also use the term "observable" in a conditional way: an effect can be observable without ever actually being observed. Although the write effect of procedure f to variable z cannot be masked, it is inconsequential in the example. Our purity analysis (Chapter 6) determines if a procedure actually observes the modification of a resource it depends on.

3.1.2 Approach

The examples in the previous section demonstrate that a procedure side-effect analysis for a higher-order, imperative program requires reasoning about control flow, value flow, and effects in that program. We therefore base our side-effect analysis on an abstract interpretation of the program that integrates control flow, value flow, and read and write effects. We extend the abstract machine from Section 2.4, which already performs flow analysis, to generate read and write effects that are caused by reading from and writing to variables and object fields (Section 3.2). Starting from an initial evaluation state, all possible successor states are explored, resulting in a flow graph annotated with effects (Section 3.2.3).

We formalize a procedure side-effect analysis for SCHEME$_0$ defined in Chapter 2. The procedure side-effect analysis is parameterized with a function *observable* that determines whether an effect is observable in a particular application context (Section 3.3). We tie a particular procedure to its side effects by collecting all side effects that are involved in all of its applications.

In this chapter, we define *observable* solely in terms of store addresses (Section 3.4). We say that an effect is observable to a caller if the address that is read or written exists in the *caller store*, i.e., the store that exists at procedure application (after evaluating procedure arguments, but before binding parameters in the environment and store). Slightly abusing terminology, we will refer to both read and write effects as "observable" when the address of the involved resource exists in the caller store.

3.1.3 Contributions

This chapter makes the following contributions:

- We extend the abstract machine for SCHEME_0 so that it tracks read and write effects generated by assigning and accessing variables and object fields during interpretation (Section 3.2).
- We develop a procedure side-effect analysis based on store addresses that takes as input flow graph annotated with effects (Section 3.3).
- We define observability of effects based on store addresses (Section 3.4).

3.2 Side-Effect Analysis

3.2.1 Design Motivation

While side-effect analysis can be fully incorporated in the abstract machine semantics, we opt for a more modular design in which the machine emits the necessary information for a subsequent effect analysis to consume. Emitting effects minimizes the changes to an abstract machine, and decouples the side-effect analysis from the actual machine. The extra effect information the abstract machine has to generate serves as an interface between machine and client effect analyses. This scheme reduces overall complexity, and enables modular reasoning over the different components that make up the resulting analysis. It also makes it possible to run certain analyses *a la carte* on the same input flow graph. We will do this in Chapters 4 and 5, where we add freshness and escape analysis to our side-effect analysis.

Side-effect information can also be recovered from a flow graph that lacks effect annotations. In the abstract machine semantics for SCHEME_0, the delta between registering effects during interpretation and deducing them from the state graph after interpretation is small. However, for more complex

languages (e.g., the semantics for JavaScript of Chapter 9) recovering these effects after interpretation can lead to a significant amount of duplicate code and coupling at design time, and redundant computation at runtime.

3.2.2 Extending the Base Semantics of SCHEME$_0$

We describe the changes that are necessary to the state space of the original abstract machine semantics for SCHEME$_0$ presented in Section 2.4.

The state space, depicted in Figure 3.1, extends the one of Figure 2.3 with effects (*Eff*). Four types of effects are tracked: reading and writing of variables (**Rv**, **Wv**), and reading and writing of object fields (**Rf**, **Wf**). They are generated by transitions between states. Effects on object fields require a field name (*FName*). SCHEME$_0$ only model the `car` and `cdr` fields of a pair. In a more general setting, the set of field names can be considerably larger or, when the abstract machine is configured for concrete semantics, infinite. For example, to add vectors to the semantics, every value that represents a legal vector index must have a counterpart in the set of field names.

Program injection and address allocation

The injection function ($\mathcal{I}$) and store address allocation functions (*allocVar*, *allocPair*) remain unchanged.

Atomic evaluation

The extended atomic evaluation function now also has to return a set of effects. Its signature therefore becomes $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup D \times \mathcal{P}(\mathit{Eff})$. If the atomic expression is a variable, then the set of effects is a singleton set containing a variable read effect (**Rv**), else the set of effects is empty.

$$\mathcal{A}(\mathit{lam}, \rho, \sigma) = (\{\mathbf{clo}(\mathit{lam}, \rho)\}, \varnothing)$$

$$\mathcal{A}(v, \rho, \sigma) = (\sigma(a), \{\mathbf{Rv}(v, a)\})$$
$$\text{where } a = \rho(v)$$

Stack address allocation

In principle, stack address allocation does not need to be modified to track side effects. However, subsequent analyses built on top of our side-effect-tracking abstract machine that treat stack addresses as application contexts,

$$
\begin{array}{rll}
\varsigma \in \mathit{State} & ::= \mathbf{ev}(e, \rho, \sigma, \iota, \kappa, \Xi) & \text{[eval state]} \\
& \mid \quad \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi) & \text{[kont state]} \\
\rho \in \mathit{Env} & = \mathsf{Var} \rightharpoonup \mathit{Addr} & \text{[environment]} \\
\sigma \in \mathit{Store} & = \mathit{Addr} \to D & \text{[store]} \\
d \in D & = \mathcal{P}(\mathit{Clo} + \mathit{Addr} + \mathit{Pair}) & \text{[value]} \\
\mathit{clo} \in \mathit{Clo} & ::= \mathbf{clo}(\mathit{lam}, \rho) & \text{[closure]} \\
p \in \mathit{Pair} & ::= \mathbf{cons}(d_{\mathrm{car}}, d_{\mathrm{cdr}}) & \text{[pair]} \\
\iota \in \mathit{LKont} & = \mathit{Frame}^* & \text{[frame]} \\
\phi \in \mathit{Frame} & ::= \mathbf{let}(v, e, \rho) & \text{[binding frame]} \\
\kappa \in \mathit{Kont} & ::= \epsilon \mid \tau & \text{[application context]} \\
\Xi \in \mathit{KStore} & = \mathit{Ctx} \to \mathcal{P}(\mathit{LKont} \times \mathit{Kont}) & \text{[stack store]} \\
\mathit{eff} \in \mathit{Eff} & ::= \mathbf{Wv}(a, v) & \text{[variable write effect]} \\
& \mid \quad \mathbf{Wf}(a, n, v) & \text{[field write effect]} \\
& \mid \quad \mathbf{Rv}(a, v) & \text{[variable read effect]} \\
& \mid \quad \mathbf{Rf}(a, n, v) & \text{[field read effect]} \\
n \in \mathit{FName} & = \{\texttt{"car"}, \texttt{"cdr"}\} & \text{[field names]} \\
E \in \mathcal{P}(\mathit{Eff}) & = \text{a set of effects} & \\
a \in \mathit{Addr} & = \text{a set of addresses} & \\
\tau \in \mathit{Ctx} & = \text{a set of stack addresses} &
\end{array}
$$

Figure 3.1: Abstract state space of the flow analysis with effects.

may impose additional requirements. For example, procedure side-effect analysis with address-based observability (Section 3.4) requires that every application context can be linked to its caller store, and procedure purity analysis (Section 6.2) requires that the procedure being applied can be determined from an application context. We revisit these assumptions when discussing those analyses in the remainder of this dissertation.

Transition relation

Side effects are produced by read and write effects that occur as a result of accessing and mutating variables and object fields during evaluation. We therefore model read and write effects explicitly on the relation that transitions between states: $(\longmapsto) \subseteq \mathit{State} \times \mathit{State} \times \mathcal{P}(\mathit{Eff})$. Every time the machine transitions and accesses or modifies a resource, it adds the appropriate effect to the set of effects for that transition. The effects of allocating or initializing variables and objects are not modeled, as they do not constitute side effects in our approach. Creation or initialization of a resource can never be observed outside the context of its creation, because the resource did not exist before entering that context. We now give the transition rules extended with effect tracking.

1. [E-ATOM] Evaluation of an atomic expression continues with the computed value and returns the set of effects that occurred during atomic evaluation.

$$\begin{aligned}\mathbf{ev}(æ, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi), E \\ \text{where } (d, E) &= \mathcal{A}(æ, \rho, \sigma)\end{aligned}$$

2. [E-LET] Evaluating a variable binding does not generate effects.

$$\mathbf{ev}(⟦\texttt{(let ((}v\ e_0\texttt{))}\ e_1\texttt{)}⟧, \rho, \sigma, \iota, \kappa, \Xi) \longmapsto \mathbf{ev}(e_0, \rho, \sigma, \mathbf{let}(v, e_1, \rho) : \iota, \kappa, \Xi), \varnothing$$

3. [E-SET] Variable mutation generates a variable write effect that is added to the set of effects resulting from atomically computing the value.

$$\begin{aligned}\mathbf{ev}(⟦\texttt{(set!}\ v\ æ\texttt{)}⟧, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto \mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), E' \\ \text{where } (d, E) &= \mathcal{A}(æ, \rho, \sigma) \\ a &= \rho(v) \\ \sigma' &= \sigma \sqcup [a \mapsto d] \\ E' &= E \cup \{\mathbf{Wv}(a, v)\}\end{aligned}$$

4. [E-APP] Evaluating a procedure call combines the effects of the atomic evaluations of operator and operand.

$$\begin{aligned}
\mathbf{ev}(\overbrace{[\![(f\ æ)]\!]}^{e}, \rho, \sigma_\tau, \iota, \kappa, \Xi) &\longmapsto \mathbf{ev}(e_0, \rho', \sigma', \langle\rangle, \tau, \Xi'), E'' \\
\text{where } (d, E) &= \mathcal{A}(f, \rho, \sigma_\tau) \\
(d_{\text{arg}}, E') &= \mathcal{A}(æ, \rho, \sigma_\tau) \\
clo &\in d \\
\mathbf{clo}([\![(\lambda(v)\, e_0)]\!], \rho'') &= clo \\
a &= allocVar(v, e, \rho, \sigma_\tau, \iota, \kappa, \Xi) \\
\rho' &= \rho''[v \mapsto a] \\
\sigma' &= \sigma_\tau \sqcup [a \mapsto d_{\text{arg}}] \\
\tau &= allocCtx(e, clo, d_{\text{arg}}, \sigma_\tau, \rho') \\
\Xi' &= \Xi \sqcup [\tau \mapsto (\iota, \kappa)] \\
E'' &= E \cup E'
\end{aligned}$$

5. [E-CONS] Allocation of a pair combines the effects of the two atomic evaluation steps for the `car` and `cdr` field.

$$\begin{aligned}
\mathbf{ev}(\overbrace{[\![(\texttt{cons}\ æ_1\ æ_2)]\!]}^{e}, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto \mathbf{ko}(\{a\}, \sigma', \iota, \kappa, \Xi), E'' \\
\text{where } (d_{\text{car}}, E) &= \mathcal{A}(æ_1, \rho, \sigma) \\
(d_{\text{cdr}}, E') &= \mathcal{A}(æ_2, \rho, \sigma) \\
a &= allocPair(e, \rho, \sigma, \iota, \kappa, \Xi) \\
\sigma' &= \sigma \sqcup [a \mapsto \{\mathbf{cons}(d_{\text{car}}, d_{\text{cdr}})\}] \\
E'' &= E \cup E'
\end{aligned}$$

6. [E-CAR] Taking the `car` field of a pair adds the appropriate field read effect to the set of effects of the atomic evaluation for obtaining the pair.

$$\begin{aligned}
\mathbf{ev}(\overbrace{[\![(\texttt{car}\ v)]\!]}^{e}, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto \mathbf{ko}(d_{\text{car}}, \sigma, \iota, \kappa, \Xi), E' \\
\text{where } (d, E) &\in \mathcal{A}(v, \rho, \sigma) \\
a &\in d \\
\mathbf{cons}(d_{\text{car}}, _) &\in \sigma(a) \\
E' &= E \cup \{\mathbf{Rf}(a, \texttt{"car"}, v)\}
\end{aligned}$$

7. [E-SET-CAR] Setting the `car` field of a pair combines the effects of the atomic evaluations of the pair and value expression with a field write effect. (The rule for setting the `cdr` is similar.)

$$\begin{aligned}\mathbf{ev}(\overbrace{[\![\texttt{(set-car! } v \text{ æ)}]\!]}^{e}, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto \mathbf{ko}(d_{\text{car}}, \sigma', \iota, \kappa, \Xi), E'' \\ \text{where } (d, E) &= \mathcal{A}(v, \rho, \sigma) \\ (d_{\text{car}}, E') &= \mathcal{A}(\text{æ}, \rho, \sigma) \\ a &\in d \\ \mathbf{cons}(\_, d_{\text{cdr}}) &\in \sigma(a) \\ \sigma' &= \sigma \sqcup [a \mapsto \mathbf{cons}(d_{\text{car}}, d_{\text{cdr}})] \\ E'' &= E \cup E' \cup \{\mathbf{Wf}(a, \texttt{"car"}, v)\}\end{aligned}$$

8. [K-RET] No effects are involved when the stack is popped on program or procedure exit.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) &\longmapsto \mathbf{ko}(d, \sigma, \iota', \kappa', \Xi), \varnothing \\ \text{where } (\iota', \kappa') &\in \Xi(\kappa)\end{aligned}$$

9. [K-LET] Because we do not consider allocation as a side effect, continuing with a `let`-created continuation involves no effects.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \mathbf{let}(v, e, \rho) : \iota, \kappa, \Xi) &\longmapsto \mathbf{ev}(e, \rho', \sigma', \iota, \kappa, \Xi), \varnothing \\ \text{where } a &= allocVar(v, e, \rho, \sigma, \iota, \kappa, \Xi) \\ \rho' &= \rho[v \mapsto a] \\ \sigma' &= \sigma \sqcup [a \mapsto d]\end{aligned}$$

3.2.3 Constructing the Annotated Flow Graph

We determine procedure side effects by reasoning about read and write effects that happen during program evaluation. We therefore construct a *flow graph* representing program evaluation, in which nodes are reachable states, and edges are transitions between states that are labeled with the effects that occur on transition. Let $\hookrightarrow$ be the transition relation $\longmapsto$ with the effects removed: $\varsigma \hookrightarrow \varsigma' \iff \varsigma \longmapsto (\varsigma', E)$. Evaluation can be expressed as computing the transitive closure of $\hookrightarrow$ after injection.

$$\mathcal{E}(e) = \{\varsigma \mid \mathcal{I}(e) \hookrightarrow^* \varsigma\}$$

The definition of flow graph G_e for expression e is then as follows:

$$\varsigma \xrightarrow{E} \varsigma' \in G_e \iff \varsigma \in \mathcal{E}(e) \text{ and } \varsigma \longmapsto (\varsigma', E)$$

In the unmodified semantics of Chapter 2 we guaranteed termination by plugging in finite sets for Var, *Addr*, and *Ctx* into the state space of the analysis. Extending the state space with effects (Figure 3.1 on page 41) adds components consisting of elements from the original state space, except for field names, which is a finite set in our semantics. Therefore the extended state space is finite under the same conditions as the original one, although when generalizing the semantics an additional condition is that the set of field names must also be finite. If these conditions are met, then the function computing the reflexive transitive closure of $\hookrightarrow$ is monotonic and therefore has a least fixpoint.

3.3 Procedure Side-Effect Analysis

Section 3.2 introduced a general side-effect analysis for SCHEME$_0$. We now shift our focus to procedures and procedure applications. The goal of procedure side-effect analysis is to compute all observable side effects that a procedure exhibits. Using the flow graph in which edges are annotated with effects, we are able to determine all side effects that occur during procedure applications. Aggregation of observable side effects has to happen along two dimensions: we have to traverse all states and their effects (Section 3.3.1), and for every effect we have to traverse the call stack (Section 3.3.2) because observability has to be determined in the context of every application context reachable on the stack.

Procedure application introduces boundaries for effects, making it possible to define a limit on a side effect's extent. Any side effect that is not observable to callers can be masked, and it is possible that a particular effect is observable in some application contexts but not in others. The goal of procedure side-effect analysis therefore is to collect all read and write effects, and for each effect and application context in which it occurs, decide whether the effect is observable. Side-effect analysis relies on a predicate $\mathit{observable} \subseteq \mathit{Eff} \times \mathit{Kont}$ that answers whether an effect *eff* in application context κ is observable or not.

The result of side-effect analysis is a mapping *se* from states to observable side effects per application context se_ς. This mapping determines which observable side effects occur in a particular application context reachable

in a particular program state. Informally speaking, procedure side-effect analysis first takes the product of all application contexts and all effects for every state, and then prunes those combinations that are not observable.

$$\begin{aligned} se \in Se = State \to Se_\varsigma \quad & \text{[state side effects]} \\ se_\varsigma \in Se_\varsigma = Kont \to \mathcal{P}(Eff) \quad & \text{[context side effects]} \end{aligned}$$

We now define our two traversal functions: one for traversing states (*travGraph*) and one for traversing stacks (*travStack*).

3.3.1 Graph traversal

We define a function $travGraph : \mathcal{P}(\varsigma) \times \mathcal{P}(\varsigma) \times Se \to Se$ that navigates flow graph G_e. The first parameter passed to *travGraph* is the set of seen states. The second parameter is the "work list", which actually is a *set* of states that still need to be visited. Function *travGraph* only needs to visit every state once. The third parameter keeps track of procedure side effects per state and context, and forms the global state of the analysis. The bottom element of the side-effect analysis domain is $(\varnothing, \{\varsigma_0\}, [])$.

Function *travGraph* has to handle three possible cases.

1. If the work list of states W is empty, the analysis has finished and returns the map of side effects.

$$travGraph(S, \varnothing, se) = se$$

2. If a state is pulled from the work list that is already in the set of seen states S, the state is removed from the work list and traversal continues.

$$travGraph(S \cup \{\varsigma\}, W \uplus \{\varsigma\}, se) = travGraph(S \cup \{\varsigma\}, W, se)$$

3. When a state ς in the work list has not yet been visited, the stack is traversed for every effect in the set of effects E associated with an outgoing transition $\varsigma \xrightarrow{E} \varsigma'$. The result of traversing the stack is a map of context side effects se_ς that associates traversed application contexts with their set of observable effects, and this mapping in turn is associated with the current state. The current state is added to the

set of seen states. The work list is updated with successor states, and graph traversal continues.

$$\begin{aligned} \mathit{travGraph}(S, W \uplus \{\varsigma\}, se) &= \mathit{travGraph}(S', W', se') \\ \text{where } se_\varsigma &= \bigsqcup \{\mathit{travStack}(\mathit{eff}, \varsigma) \mid \mathit{eff} \in E \\ &\qquad \wedge (\varsigma \xrightarrow{E} \varsigma') \in G_e\} \\ se' &= se[\varsigma \mapsto se_\varsigma] \\ W' &= W \cup \{\varsigma' \mid (\varsigma \to \varsigma') \in G_e)\} \\ S' &= S \cup \{\varsigma\} \end{aligned}$$

Procedure side-effect analysis terminates if the underlying flow graph is finite. The state and context effect maps of procedure side-effect analysis only use elements (states, application contexts, effects) from the state space of flow analysis, and each state is only visited once.

3.3.2 Stack traversal

Procedure side-effect analysis needs to determine observability of an effect for every procedure that is computing a return value at the point where the effect occurs. For this the analysis need to find all the dynamic extents in which the effect occurs, and it does so by traversing the stack. Because continuations in SCHEME_0 are delimited, local continuations are ignored entirely, and instead the analysis focuses on meta-continuations and stack addresses in the stack store.

Function *travStack* sets up stack traversal for an effect produced in a state, by delegating to function *travStack′*.

$$\begin{aligned} &\mathit{travStack} : \mathit{Eff} \times \mathit{State} \to \mathit{Se}_\varsigma \\ &\mathit{travStack}(\mathit{eff}, \varsigma) = \mathit{travStack}'(\mathit{eff}, \Xi, \varnothing, \{\kappa\}, []) \\ &\text{where } (\ldots, \kappa, \Xi) = \varsigma \end{aligned}$$

$$\mathit{travStack}' : \mathit{Eff} \times \mathit{KStore} \times \mathcal{P}(\mathit{Kont}) \times \mathcal{P}(\mathit{Kont}) \times \mathit{Se}_\varsigma \to \mathit{Se}_\varsigma$$

The first argument passed to *travStack′* is an effect. The second argument is the stack store of the state in which the effect was produced, and it is required for dereferencing underlying stack addresses. The third argument is the initially empty set of seen contexts. The fourth argument is the work list of contexts that need visiting. Stack traversal starts with the application

context on top of the stack. The fifth argument maps contexts to observable side effects, and is initially empty.

Function *travStack′* actually propagates a read or write effect down the stack and updates the side effects for reachable application contexts. If an effect is observable in an application context, then *travStack′* adds the effect to the set of observable side effects for that context. Function *travStack′* considers five distinct cases.

1. If the work list of active application contexts is empty, then stack traversal is finished, and the map of side effects is returned.

$$travStack'(\_, \_, \_, \varnothing, se_\varsigma) = se_\varsigma$$

2. An application context that was already encountered, is not processed again.

$$travStack'(eff, \Xi, S \cup \{\kappa\}, W \uplus \{\kappa\}, se_\varsigma) \\ = travStack'(eff, \Xi, S \cup \{\kappa\}, W, se_\varsigma)$$

3. The empty meta-continuation ϵ signals that the bottom of the stack has been reached. We do not treat the root (program) context as an application context, so the map se_ς is not updated.

$$travStack'(eff, \Xi, S, W \uplus \{\epsilon\}, se_\varsigma) = travStack'(eff, \Xi, S, W, se_\varsigma)$$

4. An observable effect *eff* in an application context τ that has not yet been encountered, causes the effect to be added to the set of the side effects se_ς for τ. The current application context τ is added to the set of seen contexts, and τ is dereferenced in the stack store to add all underlying application contexts to the work list.

$$travStack'(eff, \Xi, S, W \uplus \{\tau\}, se_\varsigma) = travStack'(eff, \Xi, S', W', se'_\varsigma) \\ \text{if } observable(eff, \tau) \\ \text{where } S' = S \cup \{\tau\} \\ W' = W \cup \{\kappa' \mid (\_, \kappa') \in \Xi(\tau)\} \\ se'_\varsigma = se_\varsigma \sqcup [\tau \mapsto \{eff\}]$$

5. An effect *eff* that is not observable in a previously unencountered application context τ, is masked by not adding *eff* to the set of

observable side effects for τ in se_ς. Because the effect also cannot be observed in underlying application contexts, the analysis does not need to examine underlying stacks to add contexts to the work list. The current application context τ is added to the set of seen contexts.

$$travStack'(\mathit{eff}, \Xi, S, W \uplus \{\tau\}, se_\varsigma) = travStack'(\mathit{eff}, \Xi, S', W, se_\varsigma)$$
$$\text{where } S' = S \cup \{\tau\}$$

3.4 Address-Based Observability

Section 3.3 defines a side-effect analysis in terms of observable effects, and for this reason is parameterized with function *observable*. In this section, we actually provide a definition for *observable* that can be plugged into the side-effects analysis.

At the start of this chapter we informally introduced side effects in terms of read and write effects on variables and object fields. Our abstract machine semantics represents these resources as addresses in the store. The most direct translation of our definition of side effects is therefore expressed in terms of reading and writing store addresses.

To determine whether an effect is observable or not, we need to couple every application context with its caller store, which is the store that was in effect after argument evaluation but before parameter binding. If an effect occurs on an address that is mapped in the caller store, then the effect is observable. If the address is not in the domain of the caller store, then the effect is local to the application and can be masked.

As SCHEME$_0$ adheres to ANF, operator and arguments are atomic expressions and do not mutate the store. Therefore the caller store in ANF semantics is identical to the store before evaluating the application that is involved. Either the caller store is part of the application context ($(\ldots, \sigma_\tau, \ldots) = \kappa$), as is the case in the AAC approach for example (Section 2.4.7), or we can implement a simple pre-analysis on a flow graph to collect caller stores for every context. In Section 7.5 we give an implementation of a caller-store analysis.

Address-based observability predicate $\mathit{observable}_A \subseteq \mathit{Eff} \times \mathit{Kont}$ deals with read and write effects of both variables and objects uniformly by considering only the address of the effect. If the address is mapped in the caller store of a procedure application context, then the effect is observable

from the point of view of the caller.

$$observable_A(\mathbf{Wv}(a, \_), (\ldots, \sigma_\tau, \ldots)) \iff a \in \mathrm{Dom}(\sigma_\tau)$$

$$observable_A(\mathbf{Wf}(a, \_, \_), (\ldots, \sigma_\tau, \ldots)) \iff a \in \mathrm{Dom}(\sigma_\tau)$$

$$observable_A(\mathbf{Rv}(a, \_), (\ldots, \sigma_\tau, \ldots)) \iff a \in \mathrm{Dom}(\sigma_\tau)$$

$$observable_A(\mathbf{Rf}(a, \_, \_), (\ldots, \sigma_\tau, \ldots)) \iff a \in \mathrm{Dom}(\sigma_\tau)$$

Under concrete semantics (Section 2.4.3), in which store and stack addresses are allocated with full precision, our definition of observable side effects is also fully precise, meaning that all effects that are produced during program execution are reported by the analysis, and no spurious effects are reported.

For completeness, we mention the subtle issue of idempotent writes, as in e.g. `(set! x x)`. Our analysis reports every observable write effect as such, regardless of whether the write is idempotent.

3.5 Abstract Garbage Collection

In Section 3.4 we defined address-based observability in terms of addresses that are mapped in the caller store. During the course of evaluating a program, an interpreter allocates addresses to map values in the store. As program evaluation proceeds, it may be the case that certain addresses are no longer in use and point to *garbage*. In a concrete setting (Section 2.4.3), this poses no problem with respect to precision: the store can contain any amount of garbage, because addresses are never reused and therefore truly unreachable. In an abstract setting (Section 2.4.2), however, precision can suffer because an address can be reused and become reachable again, leading to the previous value mapped at that address being joined with a new value.

Abstract garbage collection is a powerful technique for increasing the precision of flow analyses by collecting addresses that become unreachable in the course of evaluation, thereby avoiding unnecessary merging of values [Might and Shivers, 2006a]. To perform abstract garbage collection, the collector first needs to determine the set of addresses directly referenced in a certain state (the root set). Then it computes the set of all addresses reachable from that root set; all other addresses in the store are considered unreachable.

In an abstract setting, garbage collection can improve the precision of address-based observability. The definition of $observable_A$ in Section 3.4 on

page 49 checks whether an address is an element of the domain of the caller store. By removing unreachable addresses from the store, the domain of the store is kept as small as possible. This eliminates the possibility that a freshly allocated address collides with an already allocated, but no longer used address in the caller store.

Example 3.3. In the following program, procedure `f` mutates its parameter.

```
1 (define (f p)
2   (set! p 123)
3   (f 456))
4
5 (f 0)
```

Procedure `f` does not generate any observable write effects. With concrete semantics, every instantiation of parameter `p` receives a fresh, unused address. In an abstract setting in which 0CFA is used as an address allocation policy, however, every syntactic variable is its own address. This means that every instantiation of `p` (one for every application of `f`) is allocated at the same address, namely `p` itself. *Without* garbage collection, when calling `f` on line 3, the first instantiation of `p` remains mapped in the store (and therefore caller store), although at that point it has become unreachable. Therefore side-effect analysis will flag the mutation of the second instantiation of `p` on line 2 as an observable write effect, since the first instantiation of `p` is mapped in the caller store at the same address. *With* garbage collection, the first instantiation of `p` becomes unreachable at the call to `f` on line 3, and therefore will not be in the caller store. In this case, side-effect analysis will not deem the write effect generated on line 2 as observable, since the previous instantiation of `p` is not mapped in the caller store. □

3.5.1 Abstract Garbage Collection Semantics

We now define abstract garbage collection for SCHEME$_0$. Starting from a root set of addresses obtained for a particular state, all reachable addresses are computed by transitively referencing addresses in the store of that state. The store in question is then to be narrowed to the set of reachable addresses.

Referenced addresses

The overloaded function $\mathcal{T} : X \to \mathcal{P}(Addr)$ returns the set of addresses directly referenced by components in the state space.

1. The root set of an evaluation state is obtained by joining the addresses referenced by its different components. The environment is narrowed to the set of free variables in the evaluated expression.

$$\begin{aligned}\mathcal{T}(\mathbf{ev}(e, \rho, \_, \iota, \kappa, \Xi)) &= \mathcal{T}(\rho|\mathit{free}(e)) \cup \mathcal{T}(\iota) \cup \mathcal{T}_\Xi(\kappa) \\ \text{where } \mathcal{T}_\Xi(\epsilon) &= \varnothing \\ \mathcal{T}_\Xi(\tau) &= \bigcup_{(\iota,\kappa')\in\Xi(\tau)} \{\mathcal{T}(\iota) \cup \mathcal{T}_\Xi(\kappa')\}\end{aligned}$$

2. The root set of a continuation state is obtained by joining the addresses referenced by its different components. Function $\mathcal{T}_\Xi$ is the same function as defined in the preceding case for evaluation states.

$$\mathcal{T}(\mathbf{ko}(d, \_, \iota, \kappa, \Xi) = \mathcal{T}(d) \cup \mathcal{T}(\iota) \cup \mathcal{T}_\Xi(\kappa)$$

3. A closure value references the set of addresses referenced by the closure's static environment.

$$\mathcal{T}((\mathit{lam}, \rho)) = \mathcal{T}(\rho)$$

4. An address immediately references itself.

$$\mathcal{T}(a) = \{a\}$$

5. Function $\mathcal{T}$ steps into the two fields of a pair, and joins the resulting sets of referenced addresses.

$$\mathcal{T}(\mathbf{cons}(d_{\text{car}}, d_{\text{cdr}})) = \mathcal{T}(d_{\text{car}}) \cup \mathcal{T}(d_{\text{cdr}})$$

6. Environments immediately reference their range.

$$\mathcal{T}(\rho) = \text{Range}(\rho)$$

7. A `let` frame references addresses that are referenced by its environment narrowed to the set of free variables in the expression of the `let`.

$$\mathcal{T}(\mathbf{let}(\_, e, \rho)) = \mathcal{T}(\rho|\mathit{free}(e))$$

8. Function $\mathcal{T}$ recurses down a local continuation, aggregating the addresses referenced by stack frames.

$$\mathcal{T}(\phi : \iota) = \mathcal{T}(\phi) \cup \mathcal{T}(\iota)$$

9. All other components do not yield any referenced addresses.

$$\mathcal{T}(\_) = \varnothing$$

Reachable addresses

Function $\mathcal{R} : \mathcal{P}(Addr) \times \mathcal{P}(Addr) \times Store \to \mathcal{P}(Addr)$ computes the set of all addresses that are reachable from a given root set of addresses. The first parameter of $\mathcal{R}$ is the work list of addresses to be visited. The second parameter is the set of visited (and therefore reachable) addresses. The third parameter is the store in which addresses are dereferenced.

1. If the work list is empty, the set of reachable addresses is returned.

$$\mathcal{R}(\varnothing, A, \sigma) = A$$

2. An address that has already been visited, is removed from the work list without further processing.

$$\mathcal{R}(W \uplus \{a\}, A, \sigma) = \mathcal{R}(W, A, \sigma) \quad \text{if } a \in A$$

3. An address that has not yet been visited, is looked up in the store, and added to the set of seen addresses. Helper function $\mathcal{T}$ extracts the directly referenced addresses from the looked up value. The addresses are added to the work list, and processing continues.

$$\begin{aligned} \mathcal{R}(W \uplus \{a\}, A, \sigma) &= \mathcal{R}(W', A'', \sigma) \\ \text{where } A' &= \bigcup \mathcal{T}(\sigma(a)) \\ W' &= W \cup A' \\ A'' &= A \cup \{a\} \end{aligned}$$

Garbage Collector

The garbage collector $GC : State \to State$ restricts the domain of the store to the set of reachable addresses for a certain program state. We define the garbage collector for a state as follows:

$$GC(\overbrace{\mathbf{ev}(e, \rho, \sigma, \iota, \kappa, \Xi)}^{\varsigma}) = \mathbf{ev}(e, \rho|\mathit{free}(e), \sigma|A, \iota, \kappa, \Xi)$$

$$GC(\overbrace{\mathbf{ko}(d, \sigma, \iota, \kappa, \Xi)}^{\varsigma}) = \mathbf{ko}(d, \sigma|A, \iota, \kappa, \Xi)$$

$$\text{where } A = \mathcal{R}(\mathcal{T}(\varsigma), \varnothing, \sigma)$$

In this definition, A is the set of reachable addresses in state ς, and function $\mathit{free} : \mathsf{Exp} \to \mathcal{P}(\mathsf{Var})$ returns the set of free variables in an expression.

3.5.2 Extending Address-Based Observability

An address-based side-effect analysis can benefit from abstract garbage collection in two ways.

1. If the underlying flow analysis is garbage-collected, then address-based observability automatically benefits from garbage collection. This is because the definitions of $observable_A$ from Section 3.4 operate directly on the store. In a garbage-collected semantics, the exploration algorithm either garbage-collects every successor state before adding it to the work list, or every state pulled from the work list is garbage-collected before being explored.

2. In case the underlying flow analysis is not garbage-collected, the caller store must be garbage-collected before $observable_A$ from Section 3.4 is applied in order to benefit from increased precision. Address membership testing happens using the garbage-collected set of addresses $\mathcal{R}(\mathcal{T}(\varsigma), \varnothing, \sigma_\tau)$ instead of $\mathrm{Dom}(\sigma_\tau)$, where $\varsigma = \mathbf{ev}(\llbracket(f\ ae)\rrbracket, \ldots, \sigma_\tau, \ldots)$ is the program state evaluating the corresponding procedure application.

The two options require that not just the caller store, but the entire caller *state* be available. This means that adding abstract garbage collection to address observability requires some form of state-based analysis that precedes (or coincides with) side-effect analysis. We discuss how we approached this in our implementation in Section 7.5.

In Section 8.5.3 we evaluate the impact of abstract garbage collection on the precision of address-based procedure side-effect analysis.

3.6 Related Work

3.6.1 Type and Effect Systems

The majority of related work is situated in the realm of type and effect systems. Types describe the form of values that flow through programs, and type systems associate types with expressions in a program. Effect systems refine the type information by annotating the types to express further intensional or extensional properties of the semantics of the program [Nielson et al., 1999]. Effect systems can range over many such properties of a program semantics, but in the context of the discussion in this section by effects we mean the allocating, reading, and writing of store locations.

The goal of type and effect systems is to statically determine the types and effects of expressions in a program [Talpin and Jouvelot, 1992]. In terms of this goal and of traditional type and effect systems in general, the configuration of the side-effect analysis presented in this chapter with a concrete value lattice in which the elements are concrete values that are implicitly typed (see Section 7.3.1) or an abstract type lattice in which the elements are types (Section 7.3.2), constitutes a close approximation.

Type and effect systems first appear in Gifford and Lucassen [1986] in the context of integrating functional and imperative programming. While this dissertation does not explicitly state this as a goal, it shares the same philosophy by making it possible to detect and exploit side effects in functional languages extended with imperative constructs. The goal of the effect system introduced in Gifford and Lucassen [1986] is to determine effect classes for procedures. We take a similar approach when developing purity analysis on top of side-effect analysis in Chapter 6. Effects are determined only syntactically in Gifford and Lucassen's early work, and there is no support for masking unobservable effects.

Lucassen and Gifford [1988] builds upon the work of Gifford and Lucassen [1986], and introduces a static type and effect system to perform side-effect analysis. The system supports first-class procedures, effect polymorphism, and effect masking, and as such the work can be considered as seminal in its domain. Besides types and effects, the work adds a third kind of notion called a *region*, which is described as an area of the store in which side effects may occur. Regions allow the distinction between local and non-local side effects. The approach in Lucassen and Gifford [1988] relies on static declarations of effect and region parameters. A `private` expression exists to declare fresh, private regions, and it can be explicitly or implicitly (if typing allows) wrapped around expressions. To actually guarantee freshness, program states are extended with an additional component that tracks used regions. The side-effect analysis presented in this chapter does not require manual annotations, and infers everything from the original input program.

Nielson et al. [1999, Example 5] presents a side-effect analysis expressed as a type and effect system. Like our approach, they start from a simple functional language and then proceed to add imperative constructs resembling those of ML. Nielson defines a region as a set of program points at which variables could have been created, and describes effect masking as a technique to obtain sets of effects that are as small as possible. Effect masking enables removing those effects that deal with externally invisible regions. The 0CFA store address allocation policy, defined in Section 2.4.6

and used throughout this dissertation in the abstract setting, corresponds closely to this notion of regions.

The address-based instantiation of our side-effect analysis does not introduce a separate concept of regions. Instead, we reuse store and stack addresses (application contexts) from the underlying flow analysis for similar purposes as regions, namely to establish boundaries for effects and enable effect masking. Allocation is a parameter of our analysis, which enables tuning of context-sensitivity [Might and Manolios, 2009] and the use of allocators for both store addresses and (especially) stack addresses that draw from much richer sets than only syntactic sets.

Store and stack addresses are also a natural byproduct of program evaluation, and therefore no additional manual annotations or specialized constructs are required to introduce regions in our approach.

Talpin and Jouvelot [1992] presents a static approach to reconstruct maximal types and minimal effects of expressions in languages that combine functional and imperative constructs. Their work improves on previous efforts by inferring effect and region parameters instead of requiring them to be statically declared in programs.

3.6.2 Flow Analyses

This dissertation uses flow analysis computed by abstracted abstract machines as framework for approximating runtime behavior of programs with regard to effects. The advantage of our flow analysis over a type and effect system is that the precision of calling or application contexts can be configured. Using AAC or P4F as stack address allocation policy yields more control-flow precision than type-and-effect approaches that merge all calling contexts of a procedure together [Tang and Jouvelot, 1994]. Johnson and Van Horn [2014] proposes extensions to the AAC approach to flow analysis for dealing with first-class continuations, so that support for complex control-flow (e.g., through `call-with-current-continuation`) can be envisioned. A disadvantage of our approach is that we require a whole-program flow analysis, while type and effect systems support a more modular approach [Tang and Jouvelot, 1994].

Vouillon and Jouvelot [1995] formally relates type and effect systems and abstract interpretation by proving that a program complexity analysis expressed in the two frameworks are equivalent.

Talpin and Jouvelot [1992] mentions that although abstract interpretation is the "usual" framework used to approximate runtime properties of a program,

the interprocedural case "incurs a heavy computational cost". Since then, the field of abstract interpretation, especially of higher-order, functional languages, has advanced considerably (see e.g. Gilray et al. [2016] for a short overview of recent advances). Based on our experience and experiments, we find that abstract interpretation, and our particular implementation of it based on the AAC approach [Johnson and Van Horn, 2014], offers a useful tradeoff between speed and precision (see Chapter 8 and Section 9.6 for evaluation).

Similarly to Vouillon and Jouvelot [1995], Nielson et al. [1999] distinguishes between "flow based" approaches, including abstract interpretation, and "inference based" approaches, including type and effect systems. Both works stress the fact that type and effect systems are syntax-directed, while our use of abstract machine interpreters is more operational in nature and enables more precision fine-tuning.

3.6.3 Monads

Effect systems can be recasted into equivalent monad systems [Wadler and Thiemann, 2003]. However, in this setting it is unclear how to combine effects from different monads, and how to perform effect masking [Benton et al., 2002, Park and Harper, 2004].

3.7 Discussion

While we only consider variables and objects as resources, our approach can be generalized by considering consoles, printers, streams, etc. as resources. For example, `display` could generate a write effect on an abstract resource representing the current (console) output port.

We do not treat the root context as an application context, but our approach could be modified to treat this program entry context as any other application scope. The root context then can be considered to be the result of calling an implicit `main` procedure whose body is the entire program. Runtime state that persists between program runs and is modified by the program, or other outside resources (e.g., IO channels) that the program affects, generate observable side effects.

3.8 Conclusion

This chapter constructed a side-effect analysis for SCHEME_0 by extending the flow analysis from Section 2.4 with effects. The result of side-effect analysis is a flow graph representing all possible program states, and the transitions between them annotated with effects that occur on those transitions. Effects occur when resources are read from or written to. Resources are identified by store addresses, and in case of objects also field names.

If the abstract machine generates application contexts that contain the applied procedure, the annotated flow graph can be queried to determine procedure side effects. Every effect that occurs is added to all application context on the call stack.

If additionally an application context is linked to a caller store, then it becomes possible to remove unobservable side effects. Unobservable effects are effects that are not observable to callers of a particular procedure, and can be masked. An effect is observable with respect to an application context if the address of the resource involved exists in the store at the point of application. The precision of address-based observability can be increased by applying abstract garbage collection.

Chapter 4

Freshness Analysis

4.1 Introduction

In Chapter 3 we present a side-effect analysis for SCHEME$_0$ that is parameterized with a predicate *observable*. The predicate answers whether an effect is observable in a particular application context. An effect on a resource that is reachable (visible, accessible,...) by a caller is observable. If the resource exists, but is *not* reachable by callers, then it must have been created in the application context where it is used, and we call these resources *fresh*. Effects on fresh resources are unobservable by callers [Rytz et al., 2013]. In this chapter, we make these informal definitions precise when we formalize different characterizations of freshness.

We already encountered freshness of store addresses in Section 3.4, where we instantiate procedure side-effect analysis with observability based on store addresses, resulting in an *address-based* procedure side-effect analysis. In our abstract machine setting, with variables and objects as store-allocated resources, the address-based characterization of side effects corresponds closely with the notion of what exactly constitutes a side effect. However, in this chapter we show that this characterization is problematic in a static analysis setting in which resources can be allocated at addresses that are already in use. We therefore need to find and exploit other indicators for freshness, independent of store addresses.

In this chapter we show, after revisiting address freshness (Section 4.2), that locality of variables (Section 4.3) and tracking of object flow between different scopes (Section 4.4) are additional ways of determining freshness. We conclude by showing how to integrate scope-based variable and object freshness into procedure side-effect analysis (Section 4.5).

4.1.1 Contributions

- We define scope-based freshness for variables and objects (Sections 4.3 and 4.4).
- We extend procedure side-effect analysis to make use of scope-based variable and object freshness (Section 4.5).

4.2 Address Freshness

The address-based characterization of observability in Section 3.4 is attractive because it closely corresponds to our notion of side effects in the store semantics of SCHEME$_0$. Resources are represented as addresses, coupled to field names for field effects, and side effects are determined based on the addresses read from and written to. When an address is not in the caller store σ_τ of an application context τ, then that address is fresh with respect to that context. When a procedure accesses or modifies a fresh address during its application, then that side effect cannot be observed by callers of the procedure, and the effect can be masked.

$$\begin{aligned} &\mathit{freshAddress}(a, \tau) \iff a \notin \mathrm{Dom}(\sigma_\tau) \\ &\text{where } (\ldots, \sigma_\tau, \ldots) = \tau \end{aligned}$$

4.2.1 Problem: Limited Precision for Addresses

While effect masking based on addresses is attractive, it is also problematic in a typical static analysis setting.

In concrete semantics, resources that should be distinct are treated as such, because they are referenced through distinct addresses. When the abstract machine from Section 3.2 is configured with concrete value and stack store allocators, addresses are generated with full precision, and the machine allocates just like a regular interpreter for SCHEME$_0$ would at run-time. As a result, concrete side-effect analysis determines procedure side effects with full precision, i.e., with no false positives or negatives. It follows that effect masking based on addresses then also is fully precise.

However, in order to guarantee that an analysis runs in reasonable (finite) time and space, our compile-time analysis sacrifices precision primarily by *limiting the number of addresses* the abstract machine's allocator may choose from while analyzing the input program. This strategy is at the heart of

static analysis techniques for higher-order languages like AAM [Van Horn and Might, 2010], of which AAC [Johnson and Van Horn, 2014] and the semantics in this dissertation are direct descendants. For example, 0CFA is an address allocation policy for abstract semantics in which every syntactic variable is its own address (Section 2.4.6 on page 22). When effect masking hinges on addresses, we need to assess the impact of this precision loss on our approach. As it turns out, the precision loss can be problematic, as the following examples illustrate.

Example 4.1. Consider the following program.

```
1 (define (f)
2   (let ((x 1))
3     (set! x 2)
4     (f)
5     x))
```

Even though procedure `f` loops forever, it does not generate observable side effects. Side-effect analysis with address-based masking, when configured with a concrete memory allocation policy, concludes correctly that the side effect on line 3 is not observable, as the address for `x` does not exist in the caller state and therefore is fresh. This is because a concrete memory allocator *always* allocates a fresh address for variable `x` for every application of `f`. However, configured with 0CFA as an abstract memory allocation policy, side-effect analysis with address-based masking will conclude that variable `x` does exist in the caller state at the time of the second application of `f`, and therefore is unfresh. This is because variable `x` is *always* allocated at the same address. The side effect on line 3 will therefore be reported, conservatively but imprecisely, as observable. □

Example 4.2. The second example program is similar to the first one, and demonstrates that the same problem exists for object allocation.

```
1 (define (f)
2   (let ((o (cons 1 2)))
3     (set-cdr! o 3)
4     (f)
5     x))
```

The second time we enter the loop, address-based freshness analysis in combination with 0CFA considers setting the `cdr` field of object `o` on line 3 an observable side effect, although `o` is always fresh in the context of `f`. □

The examples—involving write effects, but read effects suffer the same fate—show that effect masking based solely on addresses suffers from inherent imprecision introduced by selecting addresses from a finite set. We say "inherent", because while it is always possible to recover some loss of precision by for example generating context-sensitive addresses (e.g. using 1CFA allocation [Shivers, 1991]), at one point or another the machine will run out of fresh addresses for a particular resource. Increasing context-sensitivity may cause more applications to be considered free from observable side effects by our analysis, but it takes only a single application with an observable side effect for a procedure to be marked as exhibiting side effects, and increasing context-sensitivity only delays the inevitable.

4.2.2 Solution: Scope-based Variable and Object Freshness

The precision of side-effect analysis can be improved by taking into account other characterizations of freshness than addresses or state-space components containing addresses. There are other freshness invariants that hold during concrete interpretation, which are independent of actual addresses and therefore independent of the actual choice of address allocation policy.

In the remainder of this chapter we propose two such invariants that are closely related to lexical scoping and object flow. Unlike an address-based characterization, which considers where in the *store* a resource is allocated, these invariants focus on the location in the *program* a resource is allocated.

The goal of variable and object freshness is to determine which variables and objects are fresh in a given application context. The main idea of variable and object freshness is depicted in Figure 4.1. We take the point of view of an inner scope nested inside an outer scope. Whenever a resource from an outer scope, fresh or unfresh, is accessed from (variables) or flows into (objects) a nested inner scope, then that resource is unfresh in the inner scope. This is because the resource has definitively been created before the nested scope was entered. On the other hand, whenever a resource leaves a scope, then its freshness (fresh or unfresh) is maintained. Due to lexical scoping, variables from inner scopes are not accessible, so this last point only pertains to object flow.

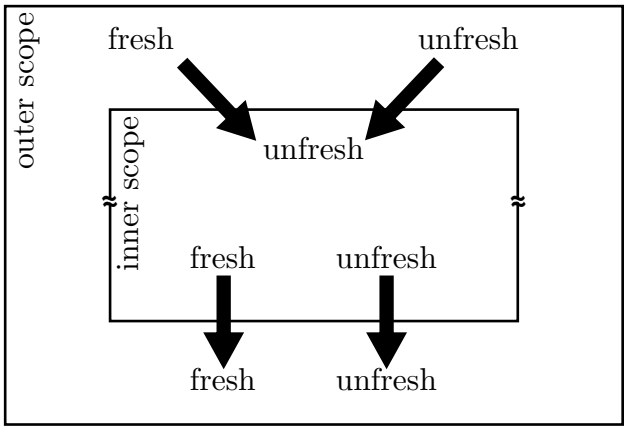


Figure 4.1: Idea behind variable and object freshness based on lexical scoping and flow.

Limitation: Topmost Application Context Only

The store and its addresses are "global" to a program state, and an address always represents the same resource (abstracted or not) when moving "down" the call stack. The same does not hold for variables, as environments in underlying stack frames may bind the same syntactic variable to different runtime copies, irrespective of the address at which the variable is allocated in the store.

Freshness analysis for variables and objects primarily operates on variables, both directly, and also indirectly because variables can reference objects. For the reasons outlined before, we can only consider variable and object freshness in the application context *on top of the stack*—i.e., the meta-continuation that is in effect in a particular program state—unless we perform additional analysis. We explain this limitation more in detail in Chapter 5, when we lift this restriction to be able to also consider freshness for underlying application contexts (Section 4.5.1).

4.3 Variable Freshness

Whenever an interpreter for SCHEME_0 enters a scope, every variable that is bound in that scope is allocated in the store. With concrete semantics, every variable is bound using a fresh (unused) address, so that different runtime environments corresponding to the same scope are guaranteed to have different bindings for these variables. However, as discussed in

Section 4.2.1, freshness of addresses typically does not carry over to a static analysis setting with finite abstract semantics. We therefore have to resort to other strategies to determine whether variables are fresh or not.

4.3.1 Examples

We show how variable freshness can be determined independently of actual addresses by giving some examples. As explained in Section 4.2.2, we only inspect freshness in the topmost application context.

Example 4.3. In the program below, variable z is bound and accessed in the scope of procedure g.

```
1 (define (g)
2   (let ((z ...))
3     z))                    ; fresh in `g`
4
5 (define (f)
6   (g))
```

This means that when reading from z on line 3, the variable is always freshly allocated in the scope of procedure g. Consequently, z is fresh when it is read during the application of procedure g. □

Example 4.4. In the program below, variable z is not bound in the scope of procedure g where it is accessed.

```
1 (define (f)
2   (let ((z ...))
3     (define (g)
4       z))                  ; not fresh in `g`
5   (g))
```

Therefore, z is allocated at some point before the call to g, and z is not fresh in g. □

4.3.2 Variable Freshness Analysis

As the examples demonstrate, considering the manner in which variables are allocated with respect to their scope already answers the question of how to determine variable freshness without looking at store addresses. A variable is fresh with respect to a scope if the variable is local to that scope.

A variable is *local* to a procedure scope if it is declared in the lexical scope established by that procedure.

Because we examine side effects at the level of procedures, we are interested in locality of variables only with respect to the lexical scope of procedures, ignoring scopes introduced by other forms like for example `let`. Additionally, variable freshness is not concerned with variables defined in inner procedure scopes, because they are not visible outside their defining scopes. We assume that an application context corresponds with an applied procedure, and therefore also to a procedure scope (Section 4.3.3). Therefore, a variable is fresh with respect to an application context if the variable is fresh in the procedure scope associated with that application context. Consequently, a free variable of a procedure is never fresh during applications of that procedure.

When a local (fresh) variable is an object reference, and the object the local variable points to is mutated, then this may or may not be visible outside that variable's scope—this is the topic of Section 4.4.

Given predicate $isLocal \subseteq \mathsf{Var} \times \mathsf{Lam}$ that answers whether a variable is local to the scope established by a procedure, then we can formalize variable freshness of a variable v in application context τ as follows.

$$\begin{aligned} &freshVariable(v, \tau) \iff isLocal(v, lam) \\ &\text{where } (\ldots, lam, \ldots) = \tau \end{aligned}$$

4.3.3 Assumptions

Freshness analysis for variables assumes that application contexts at least contain the syntactic procedure *lam* that is applied. This is necessary because our approach to variable freshness requires the scope of the applied procedure. Together with the extended environment, the applied procedure is a direct component of P4F stack addresses. It can be derived from AAC stack addresses because they contain the invoked closure. (We discussed AAC and P4F stack address allocators in Section 2.4.7 on page 23). Because a procedure corresponds to a syntactic `lambda`, and variables and their declarations are syntactic entities as well, there is no loss of precision when moving from concrete to abstract semantics.

4.4 Object Freshness

Besides variables, our side-effect analysis also treats objects as resources. Objects differ from variables in that object references are first-class values. Object references are not statically anchored by scopes like variable references are. Instead, references to objects are allowed to flow freely throughout a program. Whenever an object is allocated, a reference to that object is returned in the form of an address. This reference is a first-class value that can be stored, either by binding to variables or parameters, or assignment to variables and object fields.

With concrete semantics, every distinct object is allocated at a fresh (unused) address. This guarantee no longer exists in an abstract setting in which address can be, and typically often are, reused (Section 4.2.1). As was the case with variables, we resort once more to other strategies than inspecting an object's address for determining object freshness.

4.4.1 Informal Overview and Examples

In this section we present the ideas underlying the freshness analysis for objects through an informal overview and a series of illustrative examples. We still limit ourselves to the topmost application context (Section 4.2.2).

Informal Overview

The goal of our object freshness analysis is to determine whether expressions only reference fresh objects or not, without involving object addresses. Instead, object freshness is attached to expressions. In our abstract machine semantics of Section 2.4, `cons` is a source of fresh objects, since it creates and returns a reference to a fresh pair ([E-CONS]). Expressions such as `(set-car! (cons 1 2) 3)` do not generate observable side effects, because it is clear that expression `(cons 1 2)` only references fresh objects. However, because object references (addresses) are first-class values, it is usually no longer straightforward to determine whether an expression such as `(set-cdr! p 3)` generates an observable write effect or not, i.e., whether `p` only references fresh objects or not. This requires our analysis to propagate and update the object freshness of variables. Figure 4.1 depicts the idea of what happens when variables are bound (parameters, `let`) or assigned (`set!`). If a variable is bound or assigned to objects that exist in an outer procedure scope, then the bound or assigned variable references unfresh objects. In all other cases,

object freshness is propagated from the value expression to the bound or assigned variable. Even though the use of object freshness is limited to the topmost application context in this chapter, object freshness of variables is tracked per active application context per state. Binding the result of procedure application returning values or assignment of free variables potentially impacts the object freshness of variables in other application contexts than the topmost context.

In summary, object freshness analysis enables deciding for every expression, and variables in particular, whether they reference fresh objects or not in the topmost context of a program state.

Examples

Fresh objects are created at object allocation sites (`cons`), so this type of expression is unconditionally fresh. Object freshness is propagated throughout the program, and updated by variable binding and assignment.

Example 4.5. The following program binds local variable `o` to a fresh pair.

```
1 (define (f)
2   (let ((o (cons 1 2)))
3     (set-car! o 3)))
```

`cons` returns a freshly allocated object. Therefore, `o` references a fresh object on line 3, and no observable write effect is generated. □

When an object is referenced through a free variable, that object is considered unfresh.

Example 4.6. In this example variable `p` is a free variable of procedure `f`.

```
1 (define p ...)
2 (define (f)
3   (set-car! p 3)))
```

If variable `p` is bound to an object, that object has necessarily been allocated before the dynamic extent of the applied procedure. Therefore `p` is not fresh on line 3, and the write effect is observable to callers of `f`. □

Objects flowing to a nested procedure scope are unfresh. The two cases we have to consider are parameter binding and assignment to free variables.

Example 4.7. The next program features a field effect on a parameter.

```
1 (define (f p)
2   (set-cdr! p 3)))
```

If parameter `p` is bound to an object, that object has been necessarily allocated before the dynamic extent of the applied procedure. Therefore `p` can never point to a fresh object on line 2, and the write effect is observable to callers of `f`. □

Freshness analysis does not track individual objects flowing into procedures (as parameters) or into other objects (due to field stores). In these cases, the analysis conservatively assumes that the objects become unfresh. On procedure exit however, if the applied procedure returns a fresh value, then the procedure application itself is considered fresh.

Example 4.8. Consider the identify function `id` in the program below, which immediately returns its argument.

```
1 (define (id x)
2   x)
3 (let ((p (cons 1 2)))   ; p fresh
4   (let ((q (id p)))
5     (set-car! q 4)))    ; q unfresh
```

Even though it is clear in this example that variable `q` references a fresh object, our freshness analysis does not track object flow through the call to `id`. Instead, it considers that `id` returns parameter `q`, which points to an unfresh object. □

Example 4.9. In the program below, procedure `g` returns a fresh object.

```
1 (define (g)
2   (cons 1 2))
3 (define (f)
4   (let ((p (g)))
5     (set-cdr! p 3)))
```

Consequently, variable `p` points to a fresh object in the body of the `let`, and no observable write effect is generated on line 5. □

Even though freshness information about variables is only considered for the topmost application context, we still need to propagate freshness for variables *for every active application context* on the stack. The reason is that on procedure exit, we need to restore freshness of variables associated with an underlying application context.

Example 4.10. In the program below variable `p` is free in procedure `g` and local to procedure `f`.

```
1 (define (f)
2   (let ((p (cons 1 2)))
3     (define (g)
4       (set-car! p 3))
5     (g)
6     (set-cdr! p 4)))
```

The fact that the object of `p` is unfresh in the dynamic extent of `g` should in and by itself not impact the freshness of the same object referenced through `p` inside `f`. Therefore, after returning from the call to `g`, `p` is again a fresh object when its `cdr` field is set. □

Mutation of free variables has consequences not only in the topmost application context, but also in underlying application contexts. Because of the limitations explained in Section 4.2.2, updating freshness has to be conservative and must follow a weak update scheme. This means that object freshness of variables is monotonic in the sense that once a variable becomes unfresh in a context, it remains so.

Example 4.11. As in the previous example, variable `p` is free in procedure `g` and local to procedure `f` in the program below.

```
1 (define (f q)
2   (let ((p (cons 1 2)))
3     (define (g)
4       (set! p q))
5     (g)
6     (set-car! p 4)))
```

During the application of procedure `g`, the assignment on line 4 causes `p` to point to an unfresh object in the context of not only `g` but also `f`. □

As discussed in Section 4.3, we need to separate two notions of freshness: variable freshness, and freshness of the objects referenced by variables. It is for example possible that a fresh (local) variable points to an unfresh object, or (less likely) that an unfresh variable points to a fresh object.

4.4.2 Object Freshness Analysis

The examples illustrate that object freshness can be determined by tracking freshness from expressions in the program where freshness can be established

(allocation sites, parameters, ...) to sites where side effects occur. Object references are first-class values, and we track freshness of objects by examining variable binding and assignment. Any variable that is bound to a value originating in an outer scope, is marked as pointing to an object that is not fresh. A value that is passed around in the same scope or returned from an inner scope propagates its freshness as is.

Freshness analysis is not concerned with the actual references, including the flow of other values than object references. This means that when a variable is marked as pointing to a fresh or unfresh object, it does not actually matter whether the variable is or will ever be bound to an object. The analysis assumes that the value that flows to an expression representing the object to be mutated in `set-car!` or `set-cdr!`, is an object reference. In any case, this is checked by the runtime type system.

In the next sections we formalize object freshness analysis. We do not associate freshness with an object value itself. Freshness depends on the application context, and if we link these contexts to objects, then we are operating "after" address abstraction. Abstracted application contexts potentially map to multiple concrete contexts, and we cannot simply link object allocation to application contexts. Instead, we want to augment precision by taking into account certain concrete invariants that address abstraction destroys. Therefore, our freshness analysis more closely resembles a type system, in which freshness of referenced objects is attached to expressions, including and foremost variables.

State space

An object is either fresh ($\bot_F$) or unfresh (**unfr**). The set *Freshness* forms a join-semilattice in which $\bot_F \sqsubset \mathbf{unfr}$ and $\bot_F \sqcup \mathbf{unfr} = \mathbf{unfr}$. Variables may reference objects, and we track freshness for objects referenced by variables in a mapping from variables to freshness ($Fresh_\kappa$). Freshness of objects referenced by variables depends on the application context, so every application context has its mapping of variables to the freshness of referenced objects ($Fresh_\varsigma$). Finally, in a flow-sensitive setting, every state must be associated with freshness information for its application contexts ($Fresh$).

The state space of the analysis looks as follows.

$$\begin{aligned}
\psi \in \mathit{Freshness} &= \{\bot_F, \mathbf{unfr}\} && \text{[object freshness]}\\
F_\kappa \in \mathit{Fresh}_\kappa &= \mathsf{Var} \rightharpoonup \mathit{Freshness} && \text{[variable object freshness]}\\
F_\varsigma \in \mathit{Fresh}_\varsigma &= \mathit{Kont} \to \mathit{Fresh}_\kappa && \text{[context object freshness]}\\
F \in \mathit{Fresh} &= \mathit{State} \to \mathit{Fresh}_\varsigma && \text{[state object freshness]}
\end{aligned}$$

Atomic object freshness

Function $\mathit{fresh} : \mathsf{Atom} \times \mathit{Fresh}_\kappa \to \mathit{Freshness}$ returns the object freshness of an atomic expression.

1. A variable is looked up in the mapping from variables to freshness.

$$\mathit{fresh}(v, F_\kappa) = F_\kappa(v)$$

2. All other atomic expressions only reference fresh object.

$$\mathit{fresh}(\_, F_\kappa) = \bot_F$$

Binding

The resulting value of every expression is potentially bound to a variable using `let`. Function $\mathit{handleBinding}_F : \mathit{Freshness} \times \mathit{Fresh}_\varsigma \times \mathit{LKont} \times \mathit{Kont} \times \mathit{KStore} \to \mathit{Fresh}_\varsigma$ propagates freshness ψ of the bound expression to bound variables. Binding occurrences are represented by `let`-continuation on the stack. Function $\mathit{handleBinding}_F$ pops the stack to discover all such binding occurrences using function *pop* explained below. The object freshness of a bound variable v is updated in the application context κ' of the `let`-continuation.

$$\begin{aligned}
\mathit{handleBinding}_F(\psi, F_\varsigma, \iota, \kappa, \Xi) &= F'_\varsigma\\
&\text{where } F'_\varsigma = F_\varsigma \sqcup \bigsqcup\{[\kappa' \mapsto F'_\kappa] \mid (\iota', \kappa') \in \mathit{pop}(\iota, \kappa, \Xi)\\
&\qquad \wedge\, \mathbf{let}(v, _, _) : _ = \iota'\\
&\qquad \wedge\, F'_\kappa = F_\varsigma(\kappa') \sqcup [v \mapsto \psi]\}
\end{aligned}$$

Popping the stack

Function $\mathit{pop} : \mathit{LKont} \times \mathit{Kont} \times \mathit{KStore} \to \mathcal{P}(\mathit{LKont} \times \mathit{Kont})$ pops the stack until a non-empty local continuation is available, using auxiliary function

pop^*.

$$pop(\iota, \kappa, \Xi) = pop^*(\iota, \kappa, \Xi, \varnothing)$$

Function $pop^* : LKont \times Kont \times KStore \times \mathcal{P}(Ctx) \to \mathcal{P}(LKont \times Kont)$ guards against infinite recursion by keeping a set of seen contexts *ctxs*. This is necessary because in an abstract setting the stack may contain loops.

1. No pop occurs when the stack is empty, or when a local continuation is coupled to a context that was already encountered.

$$pop^*(\langle\rangle, \epsilon, \Xi, ctxs) = \varnothing$$

$$pop^*(\langle\rangle, \tau, \Xi, ctxs \cup \{\tau\}) = \varnothing$$

2. When popping with an empty local continuation $\langle\rangle$ and a context τ, the context is dereferenced in the stack store and added to the set of seen contexts *ctxs*.

$$pop^*(\langle\rangle, \tau, \Xi, ctxs) = \bigcup_{(\iota', \kappa') \in \Xi(\tau)} pop^*(\iota', \kappa', \Xi, ctxs \cup \{\tau\})$$

3. A pop with a local continuation returns the stack as is.

$$pop^*(\iota, \kappa, \Xi, ctxs) = \{(\iota, \kappa)\}$$

Graph traversal

Like side-effect analysis, freshness analysis extracts control flow information from the underlying flow graph of a program. It superimposes its own flow of freshness information by examining variable scope, binding, and assignment.

Function $travGraph_F : \mathcal{P}(State) \times \mathcal{P}(State) \times Fresh_\varsigma \times Fresh \to Fresh$ propagates object freshness of variables in every application context from program state to program state in a flow graph. The first two parameters of $travGraph_F$ are the set of visited states, and the work list of states to be visited. The third and fourth parameters are freshness maps: context (F_ς) and state (F) object freshness, respectively. It is F_ς that is actually updated and propagated when a state is visited, and then mapped to the state in F_ς. The bottom element of the freshness analysis domain is $(\varnothing, \{\varsigma_0\}, [], [])$. We define $travGraph_F$ in three rules.

1. If the work list of states W is empty, the analysis has finished and returns map F.

$$\mathit{travGraph}_F(\_, \varnothing, \_, F) = F$$

2. If the stated pulled from the work list is already in the set of seen states S, it is removed from the work list, and traversal continues.

$$\mathit{travGraph}_F(S \cup \{\varsigma\}, W \uplus \{\varsigma\}, F_\varsigma, F) = \mathit{travGraph}_F(S \cup \{\varsigma\}, W, F_\varsigma, F)$$

3. The analysis delegates to $\mathit{handleState}_F$ when it encounters a state it has not yet seen. The result of handling a state is a potentially updated mapping F_ς from application contexts to variable freshness for that state. If F'_ς is different from F_ς, then traversal continues with an empty set of seen states. This is to ensure that, when the analysis has finished, the analysis has visited every state with the maximum configuration of object freshness of all variables. Else, the state is added to the set of seen states. In both cases map F is updated by associating the potentially updated F'_ς with the current state. Successor states obtained from the flow graph G_e are added to the work list.

$$\begin{aligned}
\mathit{travGraph}_F(S, W \uplus \{\varsigma\}, F_\varsigma, F) &= \mathit{travGraph}_F(S', W', F'_\varsigma, F') \\
\text{where } F'_\varsigma &= \mathit{handleState}_F(\varsigma, F_\varsigma) \\
F' &= F[\varsigma \mapsto F'_\varsigma] \\
W' &= W \cup \{\varsigma' \mid (\varsigma \to \varsigma') \in G_e\} \\
S' &= \begin{cases} \varnothing & \text{if } F_\varsigma \neq F'_\varsigma \\ S \cup \{\varsigma\} & \text{else} \end{cases}
\end{aligned}$$

State Handler

Freshness information is propagated through states, and is potentially updated when an evaluation state is encountered. State handler $\mathit{handleState}_F$ dispatches on the different syntactic cases of expressions appearing in evaluation states (**ev**). We distinguish between expressions that immediately produce a value (evaluation state immediately followed by a continuation state), and expressions that the abstract machine steps into (evaluation state immediately followed by another evaluation state). The latter kind do not modify the freshness maps, because there is no value produced. Continuation states (**ko**) also do not modify the freshness maps, because we track freshness

through expressions rather than values. For the first kind of expressions, we determine the freshness based on invariants and propagated information: allocation sites return a fresh object, parameters reference an object that is unfresh, and variable freshness is looked up. Freshness is propagated when the states that immediately produce a value have a binding (or, in a more general semantics like Section 9.2.2, assignment) continuation on the top of the stack. In case the local continuation is empty, the stack is popped.

1. [E-ATOM] When a state evaluates an atomic expression, the object freshness of the atomic expression is propagated to any variable to be bound.

$$\begin{aligned} \mathit{handleState}_F(\mathbf{ev}(æ, \_, \_, \iota, \kappa, \Xi), F_\varsigma) &= F'_\varsigma \\ \text{where } F'_\varsigma &= \mathit{handleBinding}_F(\psi, F_\varsigma, \iota, \kappa, \Xi) \\ \psi &= \mathit{fresh}(æ, F_\varsigma(\kappa)) \end{aligned}$$

2. [E-SET] When a state evaluates a variable assignment, first the object freshness of the assigned variable is updated with the freshness of the assigned expression for all application contexts on the call stack, using stack-traversing function $\mathit{travStack}_F$ defined below. This results in context freshness map F'_ς. Then, because in SCHEME$_0$ semantics the result of evaluating `set!` is the assigned value itself, object freshness from this expression is propagated to every variable to be bound with this value (F''_ς).

$$\begin{aligned} \mathit{handleState}_F(\mathbf{ev}(\llbracket\texttt{(set! } v\ æ\texttt{)}\rrbracket, \_, \_, \iota, \kappa, \Xi), F_\varsigma) &= F''_\varsigma \\ \text{where } F'_\varsigma &= \mathit{travStack}_F(v, æ, \kappa, F_\varsigma) \\ F''_\varsigma &= \mathit{handleBinding}_F(\psi, F''_\varsigma, \iota, \kappa, \Xi) \\ \psi &= \mathit{fresh}(æ, F_\varsigma(\kappa)) \end{aligned}$$

3. [E-CONS] When a state evaluates `cons`, a fresh pair is returned.

$$\begin{aligned} \mathit{handleState}_F(\mathbf{ev}(\llbracket\texttt{(cons } æ_1\ æ_2\texttt{)}\rrbracket, \_, \_, \iota, \kappa, \Xi), F_\varsigma) &= F'_\varsigma \\ \text{where } F'_\varsigma &= \mathit{handleBinding}_F(\bot_F, F_\varsigma, \iota, \kappa, \Xi) \end{aligned}$$

4. [E-CAR] Object freshness analysis does not track objects flowing in and out of pairs. When a state evaluates taking the `car` field of a pair, the analysis conservatively assumes that the result is unfresh. (The case for `cdr` is similar.)

$$\begin{aligned} \mathit{handleState}_F(\mathbf{ev}(\llbracket\texttt{(car } v\texttt{)}\rrbracket, \_, \_, \iota, \kappa, \Xi), F_\varsigma) &= F'_\varsigma \\ \text{where } F'_\varsigma &= \mathit{handleBinding}_F(\mathbf{unfr}, F_\varsigma, \iota, \kappa, \Xi) \end{aligned}$$

5. [E-SET-CAR] In SCHEME$_0$ semantics, the result of a `set-car!` is the value assigned to the `"car"` field. When a state evaluates setting the `car` field of a pair, the object freshness of the value expression is propagated to variables to be bound. (The case for `set-cdr!` is similar.)

$$\begin{aligned} \mathit{handleState}_F(\mathbf{ev}(\llbracket\texttt{(set-car! } v\ æ\texttt{)}\rrbracket, \_, \_, \iota, \kappa, \Xi), F_\varsigma) &= F'_\varsigma \\ \text{where } F'_\varsigma &= \mathit{handleBinding}_F(\psi, F_\varsigma, \iota, \kappa, \Xi) \\ \psi &= \mathit{fresh}(æ, F_\varsigma(\kappa)) \end{aligned}$$

6. [E-APP] When a state evaluates an application, the state handler looks at all successor states in flow graph G_e. These successor states represent entries to all application contexts that arise from evaluating the application. For every entry state, every variable in the environment must be marked as unfresh in the entered application context. The environment at procedure entry is the static environment of a closure, which binds the free variables of the procedure, extended with parameter bindings. Therefore when a variable in the extended environment references an object, then that object has been created before the procedure application.

$$\begin{aligned} &\mathit{handleState}_F(\mathbf{ev}(\llbracket\texttt{(}f\ æ\texttt{)}\rrbracket, \ldots), F_\varsigma) = F'_\varsigma \\ &\quad \text{where } F'_\varsigma = F_\varsigma \sqcup \bigsqcup\{F'_\kappa \mid (\varsigma \to \varsigma') \in G_e \\ &\qquad\qquad \wedge\ \mathbf{ev}(\_, \rho', \_, \_, \kappa', \_) = \varsigma \\ &\qquad\qquad \wedge\ F'_\kappa = F_\varsigma(\kappa') \sqcup \bigsqcup\{[v \mapsto \mathbf{unfr}] \mid v \in \mathrm{Dom}(\rho')\}\} \end{aligned}$$

7. In all other cases, freshness map F_ς remains unchanged.

$$\mathit{handleState}_F(\_, F_\varsigma) = F_\varsigma$$

Stack Traversal

Function $\mathit{travStack}_F : \mathsf{Var} \times \mathsf{Atom} \times \mathit{Kont} \times \mathit{Fresh}_\varsigma \to \mathit{Fresh}_\varsigma$ calls $\mathit{travStack}'_F : \mathsf{Var} \times \mathsf{Atom} \times \mathcal{P}(\mathit{Kont} \times \{\mathbf{true}, \mathbf{false}\}) \times \mathcal{P}(\mathit{Kont} \times \{\mathbf{true}, \mathbf{false}\}) \to \mathit{Fresh}_\varsigma$ to initiate stack traversal with an empty set of seen contexts (third parameter) and a work list containing the topmost application context (fourth parameter). The context is tagged with a flag (top) to determine whether it can be precise in updating freshness ($\mathit{top} = \mathbf{true}$) or it should be conservative in doing so

($top = \mathbf{false}$). Traversal starts with $top = \mathbf{true}$, since freshness information can be used in the topmost application context. It is important that this flag be coupled to application contexts in both the work list and the set of seen states, because in the abstract it is possible that the same application context is on top of the stack *and* reachable as an underlying context.

$$travStack_F(v, æ, \kappa, F_\varsigma) = travStack'_F(v, æ, \varnothing, \{(\kappa, \mathbf{true})\}, F_\varsigma)$$

Because function $travStack_F$ needs to traverse all application contexts on the stack for a particular state, it differs from function *pop* in that it does not "stop" at the first non-empty local continuation. It also returns updated context freshness for all encountered application contexts instead of stacks. We describe the different cases of $travStack'_F$.

1. If the work list of active application contexts is empty, then stack traversal is finished, and the map of freshness per context for the state is returned.

$$travStack'_F(\_, \_, \_, \varnothing, F_\varsigma) = F_\varsigma$$

2. A combination of application context with a top flag that was already encountered, is not processed again.

$$travStack'_F(v, æ, S \cup \{(\kappa, top)\}, W \uplus \{(\kappa, top)\}, F_\varsigma) \\ = travStack'_F(v, æ, S \cup \{(\kappa, top)\}, W, F_\varsigma)$$

3. When the empty meta-continuation ϵ is reached, we are at the bottom of the stack. Since we do not track freshness for the global application context, only the set of seen states is extended.

$$travStack'_F(v, æ, S, W \uplus \{(\epsilon, top)\}, F_\varsigma) = travStack'_F(v, æ, S', W, F_\varsigma) \\ \text{where } S' = S \cup \{(\epsilon, top)\}$$

4. For a not previously seen combination of application context and top flag, we assign freshness of the variable in that context. If the top flag is false, then we conservatively mark the variable as unfresh in the context. Otherwise, the flag is true and the assigned variable's freshness is updated with the freshness of the atomic expression. The application context coupled to the top flag is added to the set of seen

combinations, The immediately underlying application contexts are added to the work list, coupled to the flag updated to false.

$$\begin{aligned} \mathit{travStack}'_F(v, æ, S, W \uplus \{(\tau, \mathit{top})\}, F_\varsigma) &= \mathit{travStack}'_F(v, æ, S', W', F'_\varsigma) \\ \text{where } F'_\varsigma &= F_\varsigma[\tau \mapsto F'_\kappa] \\ F'_\kappa &= F_\varsigma(\tau) \sqcup [v \mapsto \psi] \\ \psi &= \begin{cases} \mathit{fresh}(æ, F_\varsigma(\tau)) & \text{if } \mathit{top} = \mathbf{true} \\ \mathbf{unfr} & \text{else} \end{cases} \\ W' &= W \cup \{(\kappa', \mathbf{false}) \mid (\_, \kappa') \in \Xi(\tau)\} \\ S' &= S \cup \{(\tau, \mathit{top})\} \end{aligned}$$

The result of object freshness analysis can be encapsulated in a predicate *freshObject*. This predicate returns whether a variable in a stack-reachable context of a state is fresh or not. Assume that ς_0 is the initial state of a flow graph, and $F = (\varnothing, \{\varsigma_0\}, [], [])$. Then *freshObject* is defined as follows:

$$\mathit{freshObject}(v, \varsigma, \tau) \iff \mathit{fresh}(v, F(\varsigma)(\tau)) = \bot_F$$

Predicate *freshObject* only needs to handle variables, because the object that is the target of a field effect (`car`, `set-car!`, ...) is passed in as a variable.

4.5 Extending Procedure Side-Effect Analysis

Freshness can increase the precision of procedure side-effect analysis by masking unobservable side effects. Procedure side-effect analysis therefore needs to be extended to check variable freshness for effects on variables, and object freshness for effects on objects.

We equipped the procedure side-effect analysis from Section 3.3 with address-based observability (Sections 3.4 and 3.5.2). We now make the following modifications to the stack-traversing part of procedure side-effect analysis.

- We phrase observability entirely in terms of freshness, using predicates *freshAddress*, *freshVariable*, and *freshObject*. For addresses, this is just a change in perspective and adds nothing new.

- During stack traversal, we propagate a flag indicating whether we are looking at the topmost application context or not. The mechanism is similar to Section 4.5.1.

Our definition of variable and object freshness assumes that the variable involved in a variable effect, or through which a object field effect occurs, is contained in the effect. The side-effect analysis from Chapter 3 fulfills this assumption.

4.5.1 Stack traversal

As we have seen in the examples at the start of this chapter, procedure side-effect analysis needs to determine observability of an effect on every procedure that is computing a return value at the point at which the effect is produced. For this we need to find all the dynamic extents in which the effect occurs, and we do this by traversing the stack. Because continuations are delimited in SCHEME$_0$ semantics, we can ignore local continuations entirely, and instead focus on meta-continuations and stack addresses in the stack store.

We now adapt function *travStack* of procedure side-effect analysis (Section 3.3). As before, it delegates to *travStack*$'$ for the actual stack traversal, but there are two changes:

- Because function *observable*$_F$ takes a state (freshness is flow-sensitive), the entire state is passed to *travStack*$'$ instead of just the stack store.
- The work list is populated with the topmost application context tagged with a flag indicating that it is the topmost context.

$$\begin{aligned} &\mathit{travStack}(\mathit{eff}, \varsigma) = \mathit{travStack}'(\mathit{eff}, \varsigma, \varnothing, \{(\kappa, \mathbf{true})\}, []) \\ &\text{where } (\ldots, \kappa, \ldots) = \varsigma \end{aligned}$$

Function *travStack*$'$ propagates a read or write effect down the stack and updates the side effects for active application contexts, but this time the topmost flag is updated and propagated along. As in object freshness analysis, it is important that the application context and flag are coupled in the work list and the set of seen states, because in an abstract setting the topmost application context may be reachable as underlying stack context as well. Five distinct cases need to be considered.

1. If the work list of active application contexts is empty, then stack traversal is finished, and the map of side effects is returned.

$$\mathit{travStack}'(\_, \_, \_, \varnothing, se_\varsigma) = se_\varsigma$$

2. An application context in combination with a top flag that was already encountered, is not processed again.

$$\begin{aligned} &\mathit{travStack}'(\mathit{eff}, \varsigma, S \cup \{(\kappa, \mathit{top})\}, W \uplus \{(\kappa, \mathit{top})\}, se_\varsigma) \\ &\qquad = \mathit{travStack}'(\mathit{eff}, \varsigma, S \cup \{(\kappa, \mathit{top})\}, W, se_\varsigma) \end{aligned}$$

3. When the empty meta-continuation ϵ is reached, we are at the bottom of the stack. We do not treat the entry (program) context as an application context, so no further actions are taken.

$$\mathit{travStack}'(\mathit{eff}, \varsigma, S, W \uplus \{(\epsilon, \_)\}, se_\varsigma) = \mathit{travStack}'(\mathit{eff}, \varsigma, S, W, se_\varsigma)$$

4. An effect on a resource that is observable in a combination of application context τ and top flag *top* not previously seen, causes the effect to be added to the set of the side effects for τ. The application context coupled to the current flag is added to the set of seen contexts. Then τ is dereferenced in the stack store to add all underlying application contexts coupled to the updated flag (which becomes **false**) to the work list.

$$\begin{aligned} \mathit{travStack}'(\mathit{eff}, \varsigma, S, W \uplus \{(\tau, \mathit{top})\}, se_\varsigma) &= \mathit{travStack}'(\mathit{eff}, \varsigma, S', W', se_\varsigma) \\ &\qquad \text{if } \mathit{observable}_F(\mathit{eff}, \varsigma, \tau, \mathit{top}) \\ &\qquad \text{where } S' = S \cup \{(\tau, \mathit{top})\} \\ &\qquad\quad W' = W \cup \{(\kappa', \mathbf{false}) \mid (\_, \kappa') \in \Xi(\tau)\} \\ &\qquad\quad se'_\varsigma = se_\varsigma \sqcup [\tau \mapsto \{\mathit{eff}\}] \\ &\qquad\quad (\ldots, \Xi) = \varsigma \end{aligned}$$

5. An effect that is not observable in a previously unencountered combination of application context and top flag, is masked. Because the effect can never be observed by underlying application contexts, the underlying stacks do not need to be examined, and no contexts are added to the work list. The application context with flag is added to the set of seen contexts.

$$\begin{aligned} \mathit{travStack}'(\mathit{eff}, \Xi, S, W \uplus \{(\tau, \mathit{top})\}, se_\varsigma) &= \mathit{travStack}'(\mathit{eff}, \Xi, S', W, se'_\varsigma) \\ &\qquad \text{where } S' = S \cup \{(\tau, \mathit{top})\} \end{aligned}$$

4.5.2 Observability

The original signature of predicate *observable*, as defined in Section 3.3, is now extended to also receive the state and top flag as extra information. The top flag is used to determine whether variable or object freshness can be employed when deciding about observability.

Predicate $observable_F \subseteq Eff \times State \times Ctx \times \{true, false\}$ combines address freshness with variable and object freshness. From its definition it is apparent that address freshness does not depend on the position in the call stack. Observability for variable effects in the topmost application context only depends on locality for that variable with respect to the procedure being applied. For objects, we need to perform a separate object freshness analysis before running the procedure side-effect analysis with object freshness.

$$observable_F(\mathbf{Wv}(a, v), \varsigma, \tau, top) \\ \iff \neg((top = \mathbf{true} \wedge freshVariable(v, \tau)) \vee freshAddress(a, \tau))$$

$$observable_F(\mathbf{Wf}(a, n, v), \varsigma, \tau, top) \\ \iff \neg((top = \mathbf{true} \wedge freshObject(v, \varsigma, \tau)) \vee freshAddress(a, \tau))$$

$$observable_F(\mathbf{Rv}(a, v), \varsigma, \tau, top) \\ \iff \neg((top = \mathbf{true} \wedge freshVariable(v, \tau)) \vee freshAddress(a, \tau))$$

$$observable_F(\mathbf{Rf}(a, n, v), \varsigma, \tau, top) \\ \iff \neg((top = \mathbf{true} \wedge freshObject(v, \varsigma, \tau)) \vee freshAddress(a, \tau))$$

4.6 Related Work

Pearce [2011] introduces JPure, a purity analysis based on annotations that express purity, freshness, and locality. Annotation `@Fresh` is used to indicate that a method returns a fresh object, and therefore maps closely onto our notion of object freshness. Annotation `@Local` is used to indicate state owned by an object, as opposed to state only referenced by an object. If an object is fresh, and only its local state is modified, then these effects are not observable by callers. Our freshness analyses of addresses, variables, and objects also masks effects on local state. We do not explicitly differentiate

between freshness and locality, but use scope-based freshness of resources (both variables and objects) as the overarching concept.

JPure is rooted in Java, and can automatically infer annotations. Because it is in Java, it does not have to deal with variable binding and closures. JPure is meant for maintaining and inferring purity, and we revisit this work when we treat purity in Chapter 6.

The work of Rytz et al. [2013] is strongly influenced by JPure, and works with the same concepts, also expressed as annotations, but for Scala instead of Java. Their approach handles closures and higher-order procedures, but relies on explicit annotations for effects and ownership.

4.7 Conclusion

This chapter started with the observation that effects on resources that are fresh in a given application context are unobservable outside that application context. We already encountered an instance of resource freshness in Section 3.4 when defining address observability. A store address is fresh in the context of an application, if it is not mapped (or, more precisely, reachable) in the caller store.

However, relying on freshness of addresses is problematic in a typical static analysis setting. Because a static analysis has to terminate in finite space and time, the set of addresses is made finite, decreasing the precision of effect masking based on addresses. The abstract machine can run out of fresh addresses for a particular resource, which causes fresh resources to be considered as being reachable by a caller. In this chapter we therefore looked at other sources of freshness that do not directly involve addresses.

We find that certain concrete invariants concerning scope can be exploited. First, when evaluation enters a scope, variables local to that scope are fresh, irrespective of where in the store these variables are allocated. Second, certain constructs are known to be sources of fresh objects, again regardless of where in the store these objects are allocated. We can track the flow of object freshness through variables that reference objects. By exploiting these scope-based invariants, the precision of effect masking in procedure side-effect analysis can be increased (Section 8.5.3).

One important limitation of the scope-based freshness analysis presented in this chapter is that the analysis is limited to the topmost application context. We explain and lift this limitation in the next chapter. It may be considered overkill that we encoded this limitation in the framework by

propagating a flag when traversing the stack. Indeed, during procedure side-effect analysis, graph traversal initiates stack traversal and starts with the top of the stack, so any logic concerning the topmost application can be injected at this point (as we will do in Section 9.4). The reason for choosing a flag here becomes clear in the next chapter, when we adapt the update rule for this flag to allow scope-based freshness to be used in underlying application contexts.

Chapter 5

Escape Analysis

5.1 Introduction

The procedure side-effect analysis of Section 3.3 traverses the stack to determine whether a side effect is observable or not in the context of every active procedure application. Freshness was based on store addresses in Section 3.4: if the target address of an effect is not reachable in a caller store, then the address is fresh in the associated application context.

Traversing the stack is sound for addresses, because the store is "global" to each state and an address always represents the same resource, no matter in which component of a state it appears.

Chapter 4 adds scope-based variable and object freshness, but variable and object freshness analysis itself was confined to the topmost application context, with the procedure side-effect analysis only relying on address freshness when traversing the stack. The reason for this limitation is that what is true for addresses not necessarily holds for variables when traversing the stack. Every scope in a program can correspond to multiple environments during interpretation, so that the same lexical variable can have multiple runtime copies allocated in the store. With concrete semantics, every allocation of the same variable *must* have a different address. This is no longer the case when the abstract machine is configured to run with abstract semantics. Even when in the abstract variables are allocated at an identical address, they can still represent multiple concrete variables. The application contexts on the stack, even when equal in the abstract, therefore potentially correspond to different concrete environments.

The chain of environments, starting at the top of the stack, constitutes the dynamic scope. If the same variable appears more than once in the

dynamic scope, then that variable no longer uniquely identifies a runtime memory location. Traversing the different application contexts on the stack therefore means that a variable itself no longer uniquely identifies a resource.

In this chapter, we first illustrate by example that traversing the stack is also useful for freshness (Section 5.2. We then restate the problem in terms of divergence between static and dynamic scope, and give examples of cases in which stack traversal for variable and object freshness is unsound (Section 5.2.1). To guarantee that static and dynamic scope coincide, we propose to use escape analysis (Section 5.2.2). The goal of escape analysis is to determine the procedures that potentially escape their defining lexical scope. We formalize escape analysis for our semantics (Section 5.3) and extend our procedure side-effect analysis (Section 5.4) of the previous chapter with escape information to enable sound stack traversal for all types of freshness.

5.1.1 Contributions

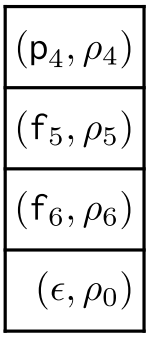
- We define an escape analysis for our semantics (Section 5.3).
- We extend side-effect analysis to make use of escape information (Section 5.4).

5.2 Motivation

Extending scope-based freshness to underlying application contexts can improve the precision of procedure side-effect analysis by masking more effects, as illustrated by the following example. The example shows that while precision-improving freshness information is available in an underlying application context, it is not accessed.

Example 5.1. Assume that the following program is evaluated with 0CFA as address allocation policy.

```
1 (define (f)
2   (let ((o (cons 1 2)))
3     (let ((g (lambda () (set-cdr! o 3))))
4       (g)
5       (f)
6       o)))
7 (f)
```



$$\rho_4 = \rho_6$$
$$\rho_5 = \rho_0[\mathsf{z} \mapsto z_5]$$
$$\rho_6 = \rho_0[\mathsf{z} \mapsto z_6]$$

Figure 5.1: Simplified depiction of the call stack and environments when executing `set!` on line 5 in example Example 5.2. Frames on the call stack are represented as tuples of the procedure that is applied with the line number as suffix, and the dynamic environment.

Procedure `f` is applied on line 7, and recursively on line 5. Address-based purity (Section 3.4) concludes that the side effect on line 3 is observable to callers of procedures `g` and `f`, because every copy of variable `o` is allocated at the same address. Topmost-only object freshness analysis (Section 4.4) does not improve address-based procedure side-effect analysis in this case. When the field write effect is generated on line 3, an application context for `g` is on top of the stack. Object freshness analysis concludes that variable `o` points to a pair that is not fresh in this application context, because `o` is a free variable in `g`. Below the application context for `g` on the stack is an application context for `f`. Freshness analysis will not step into this application context for `f`, in which `o` *does* point to a fresh object. As a result, procedure side-effect analysis concludes that `f` also produces an observable side effect, while inspection of the program shows that no side effect should escape `f`. □

5.2.1 Problem: Escaping Procedures

The example of the previous section illustrates that procedure side-effect analysis would benefit from a scope-based freshness analysis that is allowed to inspect underlying call frames. It is therefore useful to investigate why scope-based freshness analysis is limited to the top of the stack. We clarify this by discussing two programs that show that treating a variable as a unique resource identifier across the entire call stack is unsound.

The first example illustrates the problem for variable effects.

Example 5.2. Suppose that the following program is executed with concrete semantics.

```
1 (define (f p)
2   (let ((z #t))
3     (if p
4         (p)
5         (f (lambda () (set! z #f))))))
6 (f #f)
```

In the program above, procedure `f` is called twice, first on line 6 and then on line 5. A distinct copy of variable `z` is allocated for each call. During the first call to `f`, the closure created on line 5 closes over the first copy of `z`, allocated at, say, z_6. During the second call, `z` is allocated at a different address, say at z_5. Figure 5.1 depicts the call stack with associated environments at the time that the variable assignment on line 5 happens. The assignment inside the closure does not assign the copy of `z` allocated at z_5, but rather the first copy allocated at z_6. This makes z_6 an unfresh resource in the context of $(\mathtt{f}_5, \rho_5)$. Because variable `z` is local to `f`, variable freshness analysis based only on locality would therefore unsoundly consider `z` fresh in the context $(\mathtt{f}_5, \rho_5)$. □

Traversing the stack to look up a variable means that the variable is treated as having dynamic scope. The problem in Example 5.2 is that dynamic lookup of variable `z` is different from lexical or static lookup during assignment. The static lookup chain follows the extensions of static environments during application. This chain, encoded by ρ_4, can be thought of as $\rho_4 \to \rho_6 \to \rho_0$. The dynamic lookup chain, embodied by the call stack, is $\rho_4 \to \rho_5 \to \rho_6 \to \rho_0$. Note that in this example both ρ_5 and ρ_6 correspond to the procedure scope of `f`.

Example 5.2 was explained in a concrete setting. In an abstract setting with 0CFA, the problem remains. In this setting, it is the case that $z_5 = z_6$, so that $\rho_5 = \rho_6$, and contexts $(\mathtt{f}_5, \rho_5)$ and $(\mathtt{f}_6, \rho_6)$ collapse into a single abstract context $(\mathtt{f}, \rho)$ representing the first *and* the second call to `f`. Because variable `z` is local to `f`, stack-traversing variable freshness based only on locality would also unsoundly consider `z` fresh in context $(\mathtt{f}, \rho)$.

Object freshness is tracked through variable binding and assignment. Therefore the situation is similar for objects and field effects, as in the example below.

Example 5.3. Suppose again that the following program is executed with concrete semantics.

```
1 (define (f p)
2   (let ((o (cons 1 2)))
3     (if p
4         (p)
5         (f (lambda () (set-cdr! o 3))))))
6 (f #f)
```

In this example variable, `o` in the closure created at line 5 does not point to local variable `o` associated with the underlying application context, but to the local variable allocated during the first call to procedure `f`. Therefore stack-traversing object freshness analysis would unsoundly consider that `o` references a fresh object on line 5. □

In conclusion, we cannot simply traverse the stack to update or use freshness for variables and objects, because these types of freshness depend on concrete invariants concerning variables, but the same variable does not necessarily correspond to the same concrete resource when moving down the stack.

5.2.2 Solution: Escape Analysis

Variable and object freshness analysis operate on variables. Therefore, as we have shown in the examples, if during freshness analysis the stack has to be traversed, static and dynamic variable lookup must be in agreement. This is the case as long as the procedures giving rise to application contexts do not *escape* their defining lexical scope, for example by being returned from another procedure, or by being assigned to a variable. Non-escaping procedures do not flow through the program, and in principle do not need to be coupled to a static environment, but instead can just extend the dynamic (caller) environment upon application. This is why static and and dynamic variable lookup coincide when procedures do not escape. Escaping procedures are actual first-class values and need to close over the environment in which they are created.

In the remainder of this chapter we formalize an escape analysis for our semantics that maps every procedure appearing in a program onto a binary domain that tells us whether the procedure potentially escapes or not (Section 5.3). Using the results from escape analysis, we extend procedure side-effect analysis (Section 5.4) to make use of escape information when the analysis has to decide whether effects are observable or not in every application context on the call stack. As long as all procedures encountered

during stack traversal are non-escaping, freshness of variables and objects can be used. Else, the analysis has to be conservative, and fall back on address freshness for the remainder of stack traversal.

5.3 Escape analysis

The goal of escape analysis is to determine whether a procedure can escape its defining lexical scope or not. For this purpose the analysis defines a binary domain *Esc* that represents these two possibilities, and the analysis map every procedure that appears in the input program onto this domain (Ω). A non-escaping procedure is mapped onto $\bot_E$. An escaping procedure is mapped onto **dyn**.

$$esc \in Esc = \{\bot_E, \mathbf{dyn}\}$$
$$\omega \in \Omega = lam \to esc$$

The set *Esc* forms a join semi-lattice in which $\bot_E \sqcup \mathbf{dyn} = \mathbf{dyn}$.

For our purposes, we are only interested in procedures that escape their defining *procedure* scope. Therefore we do not mark a procedure that escapes a scope created by `let` as escaping.

5.3.1 Graph traversal

Escape analysis, like procedure side-effect analysis and freshness analysis, uses the flow graph computed for an input program, but escape analysis, unlike the previous analyses, is flow-insensitive. Function $travGraph_E : \mathcal{P}(State) \times \mathcal{P}(State) \times \Omega \to \Omega$ inspects all possible states, and checks all values that expressions at certain program points may evaluate to. Program points of interest are those where a closure value potentially flows outside of its defining lexical procedure scope. The first two parameters of $travGraph_E$ are a set of seen states and a work list of states. The final parameter is escape map ω, which maintains the escape status for every procedure. This map is updated for every traversed state, and the update depends on values for expressions in the state itself. The bottom element of the escape analysis domain for a program with initial state ς_0 is $(\varnothing, \{\varsigma_0\}, [])$. We define $travGraph_E$ in three rules.

1. If the work list of states W is empty, the analysis has finished and we return map ω.

$$travGraph_E(\_, \varnothing, \omega) = \omega$$

2. A state that has already been visited, is not visited again.

$$travGraph_E(S \cup \{\varsigma\}, W \uplus \{\varsigma\}, \omega) = travGraph_E(S \cup \{\varsigma\}, W, \omega)$$

3. We delegate to handler $handleState_E$ for a state in the work list, and continue traversal with the remainder of states in the work list.

$$\begin{aligned} travGraph_E(S, W \uplus \{\varsigma\}, \omega) &= travGraph_E(S', W, \omega') \\ \text{where } \omega' &= handleState_E(\varsigma, \omega) \\ S' &= S \cup \{\varsigma\} \end{aligned}$$

5.3.2 State handler

State handler $handleState_E : State \times \Omega \to \Omega$ only considers evaluation states, ignoring continuation states. For every program point in an expression through which procedures associated with closures could escape their defining lexical scope, the handler delegates to function $update_E$. Function $update_E$ checks for actual closure values flowing to an expression, and updates escape map ω accordingly.

1. For a state that evaluates an application, closures reachable from operand values are potentially escaping.

$$handleState_E(\mathbf{ev}(\llbracket(f\ æ)\rrbracket, \rho, \sigma, \_, \_, \_), \omega) = update_E(æ, \rho, \sigma, \omega)$$

2. For a state that evaluates variable mutation, closure reachable from the assigned value are potentially escaping.

$$handleState_E(\mathbf{ev}(\llbracket\texttt{(set!}\ v\ æ\texttt{)}\rrbracket, \rho, \sigma, \_, \_, \_), \omega) = update_E(æ, \rho, \sigma, \omega)$$

3. For a state that evaluates pair construction, closures from the values stored in the constructed pair are potentially escaping.

$$\begin{aligned} handleState_E(&\mathbf{ev}(\llbracket\texttt{(cons}\ æ_1\ æ_2\texttt{)}\rrbracket, \rho, \sigma, \_, \_, \_), \omega) \\ &= update_E(æ_1, \rho, \sigma, \omega) \sqcup update_E(æ_2, \rho, \sigma, \omega) \end{aligned}$$

4. For a state that evaluates assignment to the `"car"` field of a pair, closures reachable from the value expression are potentially escaping. (Similar rule for `set-cdr!`).

$$handleState_E(\mathbf{ev}(\llbracket\texttt{(set-car!}\ v\ æ\texttt{)}\rrbracket, \rho, \sigma, \_, \_, \_), \omega) = update_E(æ, \rho, \sigma, \omega)$$

5. For a state evaluating procedure exit, closures reachable from return values are potentially escaping. Only atomic expressions in return positions need to be handled. If expressions `set!`, `set-car!`, or `set-cdr!` are in a return position, then their resulting value is the value of an atomic expression that is already checked for reachable closures in one of the previous rules.

$$\mathit{handleState}_E(\mathbf{ev}(æ, \rho, \sigma, \langle\rangle, \_, \_), \omega) = \mathit{update}_E(æ, \rho, \sigma, \omega)$$

6. All other states, including evaluation of a `let` form, cannot generate procedures that escape their defining procedure scope.

$$\mathit{handleState}_E(\_, \omega) = \omega$$

5.3.3 Updating Escape Information

Function $\mathit{update}_E : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \times \Omega \to \Omega$ determines the closures that flow to a given atomic expression $æ$, and marks the associated procedures as "escaping" (**dyn**) in escape map ω. It delegates to helper function $\mathcal{T}_E$ to obtain all procedures that an atomic expression may evaluate to.

$$\mathit{update}_E(æ, \rho, \sigma, \omega) = \omega \sqcup \bigsqcup \{[\mathit{lam} \mapsto \mathbf{dyn}] \mid \mathit{lam} \in \mathcal{T}_E(d)\}$$
$$d = \mathcal{A}(æ, \rho, \sigma)$$

Referenced Procedures

Function $\mathcal{T}_E : D \to \mathcal{P}(\mathsf{Lam})$ returns the set of procedures that are directly referenced by a value.

1. A closure value references its procedure.

$$\mathcal{T}_E((\mathit{lam}, \rho)) = \{\mathit{lam}\}$$

2. No other values yield referenced procedures.

$$\mathcal{T}_E(\_) = \varnothing$$

5.4 Extending Procedure Side-Effect Analysis

Procedure side-effect analysis traverses the stack to determine the observability of an effect in every application context. Section 4.5 extended procedure side-effect analysis from Section 3.3 a first time by considering scope-based freshness of variables and objects in the topmost application context. The topmost context was distinguished from underlying application contexts by a "top" flag.

We now extend procedure side-effect analysis a second time by allowing variable and object freshness to be used in underlying application contexts as well, under the condition that no application context higher up the stack results from the application of an escaping procedure.

Instead of coupling contexts to a *top* flag indicating whether a context is the topmost context or not, we now couple application contexts to an *escape* flag which determines whether it is sound to use variable and object freshness in a particular application context. The escape flag is updated during stack traversal with the escape status ($\bot_E$ or **dyn**) of the procedure associated with the currently inspected application context each time the analysis moves to an underlying application context. Whenever the flag is equal to **dyn**, no previously encountered application context on the stack was for an escaping procedure, and scope-based freshness can be soundly used. The logic for checking this flag to decide which types of freshness can be used, is pushed down into predicate $observable_E$ (Section 5.4.2).

5.4.1 Stack Traversal

Function *travStack* sets up stack traversal by delegating to *travStack*$'$, which performs the actual traversal. Stack traversal starts with an empty set of seen contexts, the application context on top of the stack as work list, and se_ς as an empty map. The stack store is required for dereferencing underlying stack addresses.

$$\begin{aligned} &\mathit{travStack}(\mathit{eff}, \varsigma) = \mathit{travStack}'(\mathit{eff}, \varsigma, \varnothing, \{(\kappa, \bot_E)\}, []) \\ &\text{where } (\ldots, \kappa, \ldots) = \varsigma \end{aligned}$$

Function *travStack*$'$ propagates a read or write effect down the stack and updates the side effects for active application contexts. If an effect is observable in an application context, then *travStack*$'$ adds the effect to the set of observable side effects for that context. Function *travStack*$'$ considers five distinct cases.

1. If the work list of active application contexts is empty, then stack traversal is finished, and the map of side effects is returned.

$$\mathit{travStack}'(\_, \_, \_, \varnothing, se_\varsigma) = se_\varsigma$$

2. An application context in combination with an escape flag that was already encountered, is not processed again.

$$\begin{aligned}\mathit{travStack}'(\mathit{eff}, \varsigma, S \cup \{(\kappa, esc)\}, W \uplus \{(\kappa, esc)\}, se_\varsigma) \\ = \mathit{travStack}'(\mathit{eff}, \varsigma, S \cup \{(\kappa, esc)\}, W, se_\varsigma)\end{aligned}$$

3. When the empty meta-continuation ϵ is encountered, the analysis has reached the bottom of the stack. The entry (program) context is not treated as an application context, so no further action is required.

$$\mathit{travStack}'(\mathit{eff}, \varsigma, S, W \uplus \{(\epsilon, \_)\}, se_\varsigma) = \mathit{travStack}'(\mathit{eff}, \varsigma, S, W, se_\varsigma)$$

4. An effect on a resource that is observable in a combination of application context and escape flag not previously seen, causes the effect to be added to the set of the side effects for the applied procedure. The pair of application context and current flag is added to the set of seen contexts. The stack store is dereferenced to add all pairs of underlying application contexts and updated flag to the work list.

$$\begin{aligned}\mathit{travStack}'(\mathit{eff}, \varsigma, S, W \uplus \{(\tau, esc')\}, se_\varsigma) &= \mathit{travStack}'(\mathit{eff}, \varsigma, S', W', se_\varsigma) \\ &\quad \text{if } \mathit{observable}_F(\mathit{eff}, \varsigma, \tau, esc) \\ &\quad \text{where } S' = S \cup \{(\tau, esc)\} \\ &\qquad W' = W \cup \{(\kappa', esc') \mid (\_, \kappa') \in \Xi(\tau)\} \\ &\qquad se'_\varsigma = se_\varsigma \sqcup [\tau \mapsto \{\mathit{eff}\}] \\ &\qquad esc' = esc \sqcup \mathit{escapes}(lam) \\ &\qquad (\ldots, \Xi) = \varsigma\end{aligned}$$

5. An effect that is not observable in a previously unencountered combination of application context and escape flag, is masked. Because the effect can never be observable for underlying application contexts, the underlying stacks do not need to be examined, and no contexts are added to the work list. The application context with flag is added to the set of seen contexts.

$$\begin{aligned}\mathit{travStack}'(\mathit{eff}, \varsigma, S, W \uplus \{(\tau, esc)\}, se_\varsigma) &= \mathit{travStack}'(\mathit{eff}, \varsigma, S', W, se'_\varsigma) \\ &\quad \text{where } S' = S \cup \{(\tau, esc)\}\end{aligned}$$

5.4.2 Observability

In Section 4.5.2 we extended the signature of $observable_A$ with the state and top flag, and obtained $observable_F$. We now replace the top flag with the escape flag, and call the resulting function $observable_E$. The escape flag determines whether it is sound to use variable or object freshness. If the escape flag is $\bot_E$, then this means that no application context higher up the stack is the result of the application of an escaping procedure. In this case it is sound to, in addition to address freshness, use variable and object freshness. Else, the flag is **dyn**, and only address freshness can be used. Because static scope trivially agrees with dynamic scope in the topmost context, the initial value for the escape status is $\bot_E$ (i.e., it is safe to use freshness in the topmost application context).

$$observable_E(\mathbf{Wv}(a, v), \varsigma, \tau, esc) \\ \iff \neg((esc = \bot_E \wedge freshVariable(v, \tau)) \vee freshAddress(a, \tau))$$

$$observable_E(\mathbf{Wf}(a, n, v), \varsigma, \tau, esc) \\ \iff \neg((esc = \bot_E \wedge freshObject(v, \varsigma, \tau)) \vee freshAddress(a, \tau))$$

$$observable_E(\mathbf{Rv}(a, v), \varsigma, \tau, esc) \\ \iff \neg((esc = \bot_E \wedge freshVariable(v, \tau)) \vee freshAddress(a, \tau))$$

$$observable_E(\mathbf{Rf}(a, n, v), \varsigma, \tau, esc) \\ \iff \neg((esc = \bot_E \wedge freshObject(v, \varsigma, \tau)) \vee freshAddress(a, \tau))$$

5.5 Discussion

The escape analysis presented in this chapter only detects *potentially* escaping procedures. Detecting procedures that *actually* escape at call sites is more difficult. In addition to determining the flow of closure values, this also requires inspecting operator positions to verify whether the applied procedure is a procedure that escaped its defining scope. The results of our analysis therefore conservatively overapproximate the set of actual escaping procedures.

Our escape analysis is also conservative on another level. In order to keep the implementation of the abstract machine simple, certain primitive proce-

dures such as `map` are implemented in SCHEME$_0$ itself. If `map` is implemented as a primitive at the level of the interpreter, a procedure passed to `map` does not escape just because it is an argument to this primitive. However, `map` implemented as a procedure is necessarily a higher-order procedure, so that any procedure passed to it is marked as potentially escaping by our analysis.

We also experimented with extending object freshness analysis with escape analysis. In theory, this extension enables a more precise handling of mutation of free variables. However, enabling this extension did not impact the precision of procedure side-effect analysis for any of our benchmark programs (Section 8.1.3). For this reason, the object freshness analysis of Section 4.4.2 does not use the results of escape analysis.

5.6 Related Work

Escape analysis is often used as a technique for deciding which resources can be stack allocated. In an object-oriented language such as Java, objects that do not escape a method can be allocated inside the frame of an invoked method, reducing heap allocation and garbage collection overhead [Choi et al., 1999].

Functional languages, which tend to implicitly allocate many objects such as lists and closures on the heap, can use escape analysis to improve memory management by determining the lifetime of these objects [Whaley and Rinard, 1999].

To the best of our knowledge, escape analysis for Scheme was first described as a compilation pass of the Rabbit compiler [Steele Jr, 1978]. The Rabbit compiler uses escape analysis to decide whether procedures should be coupled to an environment or not. Because closures are prevalent in functional programming, it is important to choose the most efficient closure representation. Non-escaping procedures, and applications thereof, are cheaper to implement [Adams et al., 1986].

We designed our simple escape analysis along the same lines: we detect which procedures are used as first-class values, and tag them as escaping.

Other techniques exist to perform a more elaborate escape analysis that result in points-to [Whaley and Rinard, 1999] or connection [Choi et al., 1999] graphs that capture the relation between heap objects. In our approach, escape analysis is a means to an end. Therefore freshness analysis can be instantiated with (the result of) any escape analysis that is able to determine escaping procedures.

The terminology concerning escape analysis and closure analysis is not standard. In Steele Jr [1978] and other related work, escape analysis is sometimes referred to as *closure analysis*. Closure analysis in Banerjee and Schmidt [1994] is defined as "a static program analysis that approximates the set of textual lambdas that a program point can evaluate to." This definition of closure analysis still needs to be complemented with a distinct treatment of program points to arrive at our notion of escape analysis.

5.7 Conclusion

This chapter lifted the limitation of being able to use scope-based freshness of variables and objects in the topmost application context only. We presented examples that illustrate that procedure side-effect analysis can benefit from traversing the stack.

Variable and object freshness analysis from Chapter 4 rely on variables. Reasoning about variables in conjunction with stack traversal can be problematic when procedures escape, because the dynamic lookup chain represented by the call stack does not necessarily agree with the static lookup of variables. Escape analysis enables us to determine which procedures potentially escape, and therefore under which circumstances variables can be assumed to represent the same resources when traversing the stack.

We adapted procedure side-effect analysis by replacing the top flag with an escape flag. The escape flag keeps track of whether procedures higher on the stack might have escaped when reaching a certain application context during stack traversal. Escape information renders procedure side-effect analysis more precise by allowing variable and object freshness to be used in application contexts reachable from the topmost calling-context, thereby possibly masking more effects involving free variables than address freshness alone (Section 8.5.3).

Chapter 6

Purity Analysis

> Functions and routines are as different in their nature as expressions and commands. It is unfortunate, therefore, that most programming languages manage to confuse them very successfully.
>
> — Christopher Strachey

6.1 Introduction

Mathematically speaking, the only effect of a function is turning input arguments into a resulting value that only depends on those input arguments. However, in most programming languages callable entities (procedure, method, function, constructor, subroutine, ...) can do more than that, even in programming languages that de-emphasize assignment (e.g. Scheme, Scala), and even in languages that refer to these entities as "functions" (e.g., FORTRAN, Erlang, JavaScript). Anything else a procedure does besides producing a value, is called a *side effect* of that procedure.

Side effects impact the purity of a procedure. When the procedure modifies a resource reachable by the caller, the procedure *generates* an observable side effect. Side effects external to the procedure also play a role. When a procedure accesses a program resource other than one of its input arguments, and this resource is modified in between applications of that procedure, then that procedure *depends* on an external side effect.

We can summarize the behavior of a procedure w.r.t. side effects occurring in the program. A procedure can:

- generate a side effect,
- depend on an external side effect,
- or both,
- or none of the above.

Based on the side-effecting behavior of procedures, different classifications of purity are possible. The strictest (mathematical) definition of purity considers a procedure as *pure* if it does not generate observable side effects *and* it does not depend on external side effects. A weaker notion of purity considers a procedure pure if it does not generate observable side effects, i.e., it is allowed to depend on external side effects. Procedures that allocate and mutate memory locations, or that depend on mutable memory locations, can still be considered pure when these effects can not be observed. Additionally, in our approach pure procedures may return procedures and objects allocated by those procedures.

In this chapter we present an approach to determining procedure purity in SCHEME$_0$. Our purity analysis builds on the side-effect analysis and observability from Chapters 3 to 5. Procedure purity is linked to the side effects that occur during and in between procedure applications:

- *Procedure side-effect analysis* of Section 3.3, and its refinements in Sections 4.5 and 5.4, is used to determine all observable read and write effects of procedures.
- Regular *side-effect analysis* of Section 3.2 is used to identify write effects in between applications, regardless of the actual context in which they occur.

As with procedure side effects, the behavior of all the applications of a procedure determines overall procedure purity. For example, we consider a procedure pure only if all its applications are pure. We develop a formal definition of purity in Section 6.2.

6.1.1 Terminology

A procedure is not a function in the mathematical sense if it either generates side effects, or depends on external side effects, or both. However, the term "function" is widely used both in the literature and by programmers to designate a callable entity that contains executable code, whether the code is allowed to produce or depend on side effects or not. As mentioned before, some languages use the terminology, or even the syntactic `function` keyword, to designate callable entities, again irrespective of any side effects that may or may not be present. As a reminder, in the context of SCHEME$_0$ we use the term *procedure* to signify a syntactic occurrence of the `lambda` special form in a program, and prefer the use of the term *closure* to designate a `lambda` expression coupled to its static environment.

6.1.2 Motivation

Side effects make programs more difficult to understand, as they prevent one from treating procedures as only mappings from input to output. This means that we lose the ability to abstract procedure calls into resulting values, rendering programs more difficult to understand for both humans and tools. Purity as a property of procedures appearing in a program constrains these procedures into the mathematical meaning of functions.

Research in different areas has demonstrated that purity information aids program understanding, specification, testing, debugging, and maintenance [Finifter et al., 2008]. Therefore, determining or verifying purity is useful for software engineering purposes.

Security Purity facilitates establishing security and confidentiality aspects of applications [Barnett et al., 2006]. Pure procedures are more secure than impure ones, because they interfere less with the rest of the application. Pure procedures only communicate with the rest of the program through their return values, and otherwise are not able to leak information though side-channels [Finifter et al., 2008].

Bug reduction Purity also has the potential to reduce the number of bugs in programs, and makes it easier to reproduce bugs [Hughes, 1989]. Because they present less of a mental burden for programmers than procedures with observable side effects, pure procedures are usually easier to reason about in isolation. When things do go wrong it is possible to call or "restart" a pure procedure with the arguments

causing unwanted behavior, without having to reset or recreate external program state. Determinism of pure procedures ensures that, given the same argument values, these procedures will behave in exactly the same way, making it more straightforward to reproduce and solve bugs.

Assertions and contracts Pure procedures can be safely called from assertions and contracts, thereby augmenting the expressivity of these constructs [Jacobs et al., 2011]. Assertions and contracts should not be allowed to impact a program of which they are guarding some specific behavior. Purity can help rule out unwanted behavior and guarantee transparent semantics of any executable program annotation.

Modularity Observable procedure side effects introduce dependence on resources outside those procedures. Therefore it is often necessary to enforce purity in order to ensure reuse.

Program optimization Compilers and runtimes can use purity to optimize program performance [Rytz et al., 2013]. Pure expressions relax ordering constraints for code generators, thereby enabling several optimizations such as parallelization, memoization, and laziness [Nicolay et al., 2011, Pitidis and Sagonas, 2011, Shirako et al., 2007].

As Finifter et al. [2008] argues, security and privacy requirements of real-world programs are often expressed in terms of reproducibility, invertibility, non-interference, or containment of untrusted code. Determining the purity of certain procedures helps in verifying these security properties. The authors then give the example of electronic voting software, in which presenting voting information and manipulating a voter's actions should be performed by pure procedures. The absence of side effects during a voting session ensures that no sensitive information is leaked for example. Ensuring that the voting session is independent of effects caused by previous voters' interactions makes sure that voter privacy is respected and malicious interference from the outside is avoided. Finifter et al. [2008] concludes by stating that verifying whether a computation is deterministic and free of side effects is a difficult task that typically requires careful examination of a program's entire source code. The authors mention that this task is especially challenging in imperative languages commonly used to write real-world programs, and which permit side effects and data dependencies that are difficult to reason about.

Besides security and the other properties we already mentioned, proving the absence of (observable) side effects has other advantages and applications,

which are discussed more extensively in related work on purity and side-effect analysis (Section 6.5).

Although purity is an important step in determining whether expressions and procedures can be regarded as safe and deterministic, it is in principle not sufficient. Other non-functional factors, like non-termination and resource starvation also play a role, but are out of the scope of this dissertation.

6.1.3 Examples

Purity analysis for procedures involves determining how a procedure behaves in terms of side effects with respect to the rest of the program. Purity analysis relies on general side-effect analysis, and on procedure side-effect analysis to compute observable side effects of procedures at compile-time. We revisit some of the examples from procedure side-effect analysis (Chapter 3 and recast them into a purity analysis setting.

Because the purity of a procedure is determined by side effects occurring during its applications, every side effect that occurs is examined in every application context on the call stack.

Example 6.1. The example program below is identical to the program in Example 3.1 on page 37.

```
1 (define (f)                ; pure
2   (define (g p)            ; generates side effect
3     (h p))
4   (define (h q)            ; generates side effect
5     (set-car! q 3))
6   (let ((o (cons 1 2)))
7     (g o)))
8 (f)
```

Procedure `f` creates a pair and passes this object to procedure `g`, which in turn passes it to procedure `h`. When function `h` modifies the `car` field of the pair, this results in a write effect for procedures `h` and `g` that is observable by their callers on lines 3 and 7, respectively. Because procedures `g` and `h` generate an observable side effect, they are not pure. However, the write effect is not observable to callers of procedure `f`. Because procedure `f` does not depend on external side effects, `f` is a pure procedure. □

It is not sufficient to examine effects in the context of every active application. Write effects in between applications, independent of the context

in which they appear, also play a role. Our purity analysis reasons about *generating* observable effects on the level of variables or objects. Example 6.2 illustrates that field names are also taken into account when the analysis reasons about *dependence* on external side effects and objects are involved.

Example 6.2. Consider the following program, in which procedure `f` reads from the pair referenced by variable `z`.

```
1 (let ((z (cons 1 2)))
2   (define (f)          ; depends on external side effect
3     (cdr z))
4   (f)
5   (set-cdr! z 3)
6   (f))
```

Procedure `f` is applied twice, with a mutation of the `cdr` field in between the two applications. Procedure `f` is impure because its resulting value depends on the value of an object field that is mutated during program execution. If in the above program `f` would read the `car` field on line 3, or the `car` field would be mutated on line 5, then `f` would not depend on an external side effect. □

As a final example, we look at a higher-order procedure.

Example 6.3. We revisit the program from Example 3.2 on page 37.

```
1 (define (f h)       ; generates observable write effect
2   (h))
3 (let ((z #t))
4   (define (g)       ; generates observable write effect
5     (set! z #f))
6   (f g))
```

Procedure `g` is impure because it mutates its free variable `z`. Procedure `f` receives procedure `g` as argument and invokes it. Therefore, the write effect of `g` on `z` is also visible to callers of `f`, making procedure `f` also impure. □

6.1.4 Approach

We base our purity analysis on the side-effect analysis of Section 3.2, which is capable of identifying all side effects that occur during and in between procedure applications, and on the procedure side-effect analysis of Section 3.3 and its refinements in later chapters, which is able to determine the observability of those effects with respect to application contexts.

$$
\begin{aligned}
\overline{\mathit{eff}} \in \overline{\mathit{Eff}} &= \{\mathbf{gen}, \mathbf{obs}\} && \text{[side-effecting behavior]} \\
\overline{E} \in \mathcal{P}(\overline{\mathit{Eff}}) &= \text{a set of side-effecting behaviors} \\
P \in \mathit{Purity} &= \mathsf{Lam} \rightharpoonup \mathcal{P}(\overline{\mathit{Eff}}) && \text{[procedure purity]} \\
R \in \mathit{Read} &= \mathit{Res} \rightharpoonup \mathcal{P}(\mathsf{Lam}) && \text{[read table]} \\
O \in \mathit{Obs} &= \mathit{Res} \rightharpoonup \mathcal{P}(\mathsf{Lam}) && \text{[observer table]} \\
\mathit{res} \in \mathit{Res} &= \mathit{Addr} + (\mathit{Addr} \times \mathit{FName}) && \text{[resource]}
\end{aligned}
$$

Figure 6.1: State-space of puirty analysis.

Purity analysis examines the side effects that occur in a program by traversing a flow graph of that program annotated with effects, and by traversing the call stack that is in effect in states. The analysis summarizes the *side-effecting behavior* of each procedure, i.e., how each procedure behaves during program execution in terms of side effects (Section 6.2). Purity analysis considers two types of side-effecting behavior: generating observable side effects, and depending on external side effects. The result of purity analysis is a mapping from procedures to side-effecting behavior. We can further classify procedures by mapping effect summaries to specific effect classes (Section 6.3), thereby supporting different definitions of purity that exist in the literature. For example, to determine whether a procedure is pure in the mathematical sense, we consider all applications of that procedure, and if for any such application it generates no observable side effects, and does not depend on external side effects, then the procedure is declared pure.

6.1.5 Contributions

This chapter makes the following contributions:

- A purity analysis that determines whether procedures generate observable side effects or depend on external side effects (Section 6.2).

- A classification of side-effecting behavior according to different definitions of purity found in related work (Section 6.3).

6.2 Purity Analysis

Figure 6.1 depicts the state-space for purity analysis. The goal of purity analysis is to map each procedure onto a set of side-effecting behaviors in $\overline{Eff}$. Purity analysis requires two sources of side effects. First, it uses a flow graph in which edges are annotated with effects to be able to determine all side effects that occur during and in between procedure applications in a flow-sensitive manner (Section 3.2). Second, it uses the result of procedure side-effect analysis to determine which of these side effects are observable in the context of active procedure applications (Section 3.3). Purity analysis examines all side effect information, and summarizes the side-effecting behavior of each procedure (*Purity*). There are two types of behavior we are interested in.

Generation of side effects When a procedure generates an observable write effect, **gen** is added to the set of side-effecting behaviors of that procedure.

Dependence on external side effects Side-effecting behavior **obs** for a procedure signifies that the procedure depends on an external side effect. To be able to determine this kind of dependence, purity analysis employs two tables. A read table (*Read*) tracks the procedures that have a read dependency on a resource. An observer tables (*Obs*) tracks the procedures that become observers upon a read of a resource. For dependence on external effects, purity analysis looks for read–write–read sequences involving the same resource (for reading and writing) and the same procedure (for the initial and subsequent read). If a procedure reads an external resource, this is tracked in the read table. If that resource is mutated, then all procedures that have a read dependency on that resource are added to the observer table as potential observers. If a procedure reads a resource, and is a potential observer for that resource, then a read–write–read sequence is completed, and that procedure becomes an actual observer of an external effect. If this is the case, then **obs** is added to the procedure's set of side-effecting behaviors.

While *Eff* is the set of effects (Figure 3.1), we denote the set that summarizes the side-effecting behavior of procedures as $\overline{Eff}$. A resource is either an address of a variable, or an address of an object together with a field name.

As a brief reminder, we captured side-effect information in a structure Se that enables us to retrieve all observable effects in application contexts reachable in a certain state (Section 3.3).

$$\begin{aligned} se \in Se = State \to Se_\varsigma \quad & \text{[state side effects]} \\ se_\varsigma \in Se_\varsigma = Kont \to \mathcal{P}(Eff) \quad & \text{[context side effects]} \end{aligned}$$

6.2.1 Graph traversal

We define a function $travGraph_P$ that navigates flow graph G_e and propagates information about effects per procedure.

$$travGraph_P : Se \times \mathcal{P}(State) \times \mathcal{P}(State) \times Purity \times Read \times Obs \to Purity$$

The first parameter of this function are the state side effects, computed by procedure side-effect analysis (Section 3.3). It is a mapping from states to side effects per active application context in a state. The second and third parameter are the set of seen states and the work list, respectively. The final three parameters are the purity maps of the analysis, namely P, R, and O. The bottom element of purity analysis domain is $(se, \varnothing, \{\varsigma_0\}, [], [], [])$. Function $travGraph_P$ has to handle three possible cases.

1. If the work list of states W is empty, the analysis terminates and the map of side-effecting behaviors per procedure P is returned.

$$travGraph_P(\_, \_, \varnothing, P, \_, \_) = P$$

2. If a state in the work list is already in the set of seen states S, it is removed from the work list and traversal continues.

$$\begin{aligned} &travGraph_P(se, S \cup \{\varsigma\}, W \uplus \{\varsigma\}, P, R, O) \\ &\qquad = travGraph_P(se, S \cup \{\varsigma\}, W, P, R, O) \end{aligned}$$

3. The interesting case is when the analysis encounters a state that it has not yet seen in the current purity maps. For every effect in the set of effects E associated with an outgoing transition, the analysis delegates to a handler for that effect. Besides the actual effect and the purity maps of the analysis, the effect handler also receives a mapping from all active contexts to their set of observable side effects se_ς. The handler returns potentially updated purity maps, and if any of these

maps increases, the set of seen states is cleared to make sure that at the end of graph traversal the analysis has visited each state with the maximum attainable purity maps. If the purity maps remains unchanged, the current state is added to the set of seen states. The work list is then updated with successor states, and traversal continues.

$$
\begin{aligned}
&\mathit{travGraph}_P(se, S, W \uplus \{\varsigma\}, P, R, O) = \mathit{travGraph}_P(se, S', W', P', R', O') \\
&\qquad \text{where } (P', R', O') = \bigsqcup \{\mathit{handleEffect}(\mathit{eff}, se_\varsigma, P, R, O) \mid \\
&\qquad\qquad \mathit{eff} \in E \wedge (\varsigma \xrightarrow{E} \varsigma') \in G_e\} \\
&\qquad\qquad se_\varsigma = se(\varsigma) \\
&\qquad\qquad W' = W \cup \{\varsigma' \mid (\varsigma \to \varsigma') \in G_e)\} \\
&\qquad\qquad S' = \begin{cases} \varnothing & \text{if } P \neq P' \vee R \neq R' \vee O \neq O' \\ S \cup \{\varsigma\} & \text{else} \end{cases}
\end{aligned}
$$

The analysis is finite if the underlying flow graph is finite, because the different maps of the analysis monotonically increase with values chosen from finite sets. Procedures that are mapped onto $\bot$ in the resulting behavior map are procedures that never were applied during abstract interpretation.

6.2.2 Effect handlers

Function *handleEffect* is an intermediate dispatcher that forwards to *handleRead* and *handleWrite*.

$$\mathit{handleEffect} : \mathit{Eff} \times \mathit{Se}_\varsigma \times \mathit{Purity} \times \mathit{Read} \times \mathit{Obs} \to \mathit{Purity} \times \mathit{Read} \times \mathit{Obs}$$

The first parameter of this function is an effect *eff* that occurs in a particular state. The second parameter is the set of observable side effects per active application context se_ς for the same state in which the effect occurs. In case of effects on objects, the function combines the address of the object and the

field name into a resource, else the resource is just the address of a variable.

$$\begin{aligned} &\mathit{handleEffect}(\mathbf{Wv}(a, \_), se_\varsigma, P, R, O) \\ &\qquad = \mathit{handleWrite}(\mathit{eff}, a, se_\varsigma, P, R, O) \end{aligned}$$

$$\begin{aligned} &\mathit{handleEffect}(\mathbf{Wf}(a, n, \_), se_\varsigma, P, R, O) \\ &\qquad = \mathit{handleWrite}(\mathit{eff}, (a, n), se_\varsigma, P, R, O) \end{aligned}$$

$$\begin{aligned} &\mathit{handleEffect}(\mathbf{Rv}(a, \_), se_\varsigma, P, R, O) \\ &\qquad = \mathit{handleRead}(\mathit{eff}, a, se_\varsigma, P, R, O) \end{aligned}$$

$$\begin{aligned} &\mathit{handleEffect}(\mathbf{Rf}(a, n, \_), se_\varsigma, P, R, O) \\ &\qquad = \mathit{handleRead}(\mathit{eff}, (a, n), se_\varsigma, P, R, O) \end{aligned}$$

Handling a read potentially updates the side-effecting behavior for a procedure and the read table. The effect handler *handleRead* delegates to helper function $\mathit{travStack}_{PR}$ to inspect all active procedure applications.

$$\begin{aligned} &\mathit{handleRead}(\mathit{eff}, \mathit{res}, se_\varsigma, P, R, O) = (P', R', O) \\ &\qquad \text{where } (P', R') = \mathit{travStack}_{PR}(\mathit{eff}, \mathit{res}, se_\varsigma, P, R, O) \end{aligned}$$

Handling a write potentially updates the side-effecting behavior for active procedures and the observer table for the affected resource. The effect handler *handleWrite* delegates to helper function $\mathit{travStack}_{PW}$ to inspect all active procedure applications.

$$\begin{aligned} &\mathit{handleWrite}(\mathit{eff}, \mathit{res}, se_\varsigma, P, R, O) = (P', R, O') \\ &\qquad \text{where } O' = \mathit{updateObs}(\mathit{res}, R, O) \\ &\qquad\qquad P' = \mathit{travStack}_{PW}(\mathit{eff}, \mathit{res}, se_\varsigma, P) \end{aligned}$$

Updating observers does not depend on the observability of the write effect, i.e., it does not matter where and how the write effect occurs. If a resource is written, then every procedure that has a read dependency on that resource is unconditionally moved to the observer table. Every procedure that has a read dependency on the written resource, is added to the observer table under that same resource, regardless of whether the write effect is observable or not.

$$\begin{aligned} &\mathit{updateObs}(\mathit{res}, R, O) = O' \\ &\qquad \text{where } O' = O \sqcup \bigsqcup \{[\mathit{res} \mapsto \{\mathit{lam}\}] \mid \mathit{lam} \in R(\mathit{res})\} \end{aligned}$$

The following example illustrates the unconditionality of the write effect in a read-write-read sequence.

Example 6.4. Procedure `g` reads variable `z` that procedure `f` mutates in between calls to `g`.

```
1 (define (f)
2   (define z #f)
3   (define (g)
4     z)
5   (g)
6   (set! z #t)
7   (g))
8 (f)
```

In the application context of `f`, all read effects (line 4) and the write effect (line 6) on `z` are local and can be masked. However, for detecting dependence of procedure `g` on external side effects, it does not matter that the write effect can be masked in the context of a call to `f`. The write effect has to be taken into account to determine that `g` depends on this write to `z`. □

6.2.3 Stack traversal

Effects impact all procedures on the call stack. Therefore, like procedure side-effect analysis, purity analysis must traverse the stack to find all active procedure applications in a given state, and determine the observable effects they generate for each application. For this purpose, we parameterized purity analysis with the result from side-effect analysis.

Instead of doing a proper stack traversal of its own, purity analysis can, for a particular state, reuse the information already computed by procedure side-effect analysis for that state. Purity analysis can inspect the domain of the mapping from contexts to the set of observable effects. We define function $procsObservable : Eff \times Se \to \mathcal{P}(\mathsf{Lam})$ that, given an effect, crawls the side-effect information of a state and returns the set of procedures for which this effect is observable.

$$\begin{aligned} procsObservable(eff, se_\varsigma) = \{lam \mid\ & \tau \in \mathrm{Dom}(se_\varsigma) \\ & \wedge\ se_\varsigma(\tau) = E \\ & \wedge\ eff \in E \\ & \wedge\ (\ldots, lam, \ldots) = \tau\} \end{aligned}$$

Now that we have our implicit stack traversal encapsulated by *procsObservable*, we explain how to propagate read and write effects down the stack.

Read effect

Propagating a read effect down the stack potentially marks active procedures as observers, and checks whether the read table needs to be updated for the read resource. If the read effect is observable in an application context on the stack, and the applied procedure is registered in the observer table for the read resource, then **obs** is added to the set of side-effecting behaviors of that procedure. Additionally, if the read effect is observable in an application context on the stack, then a read dependency is registered in the read table for the applied procedure.

$$
\begin{aligned}
&\mathit{travStack}_{PR}(\mathit{eff}, \mathit{res}, \mathit{se}_\varsigma, P, R, O) = (P', R') \\
&\quad \text{where } P' = P \sqcup \bigsqcup \{[\mathit{lam} \mapsto \{\mathbf{obs}\}] \mid \mathit{lam} \in O(\mathit{res}) \\
&\qquad\qquad \wedge\, \mathit{lam} \in \mathit{procsObservable}(\mathit{eff}, \mathit{se}_\varsigma)\} \\
&\qquad R' = R \sqcup \bigsqcup \{[\mathit{res} \mapsto \{\mathit{lam}\}] \mid \mathit{lam} \in \mathit{procsObservable}(\mathit{eff}, \mathit{se}_\varsigma)\}
\end{aligned}
$$

Write effect

Propagating a write effect down the stack potentially marks active procedures as generating an observable write effect. If the write effect is observable in an application context on the stack, then **gen** is added to the set of side-effecting behaviors of the applied procedure.

$$
\begin{aligned}
&\mathit{travStack}_{PW}(\mathit{eff}, \mathit{res}, \mathit{se}_\varsigma, P, R, O) = P' \\
&\quad \text{where } P' = P \sqcup \bigsqcup \{[\mathit{lam} \mapsto \{\mathbf{gen}\}] \mid \mathit{lam} \in \mathit{procsObservable}(\mathit{eff}, \mathit{se}_\varsigma)\}
\end{aligned}
$$

6.3 Purity Classification

The side-effecting behavior of a procedure (**gen** and **obs**) can be mapped to an effect class for the procedure, depending on the definition of purity.

$$
\begin{aligned}
\mathit{class} \in \mathit{Class} &= \text{a set of effect classes} \\
\mathit{classify} : \mathcal{P}(\overline{\mathit{Eff}}) &\to \mathit{Class} \\
P \in \mathit{Purity} &= \mathsf{Lam} \to \mathit{Class}
\end{aligned}
$$

The related work on purity can be divided into two categories: those that take dependence on external effects into account, and those that do not. The majority of related work [Huang et al., 2012, Madhavan et al., 2011, Pearce, 2011, Rytz et al., 2013, Salcianu and Rinard, 2005] considers a procedure to be pure if it does not generate observable write effects.

$$\mathit{Class} = \{\mathbf{pure}, \mathbf{impure}\}$$
$$\mathit{classify}(\overline{E}) = \begin{cases} \mathbf{impure} & \text{if } \mathbf{gen} \in \overline{E} \\ \mathbf{pure} & \text{else} \end{cases}$$

Our definition of purity, and of other related work [Finifter et al., 2008, Pitidis and Sagonas, 2011], considers a procedure pure if additionally it does not depend on external side effects.

$$\mathit{Class} = \{\mathbf{pure}, \mathbf{impure}\}$$
$$\mathit{classify}(\overline{E}) = \begin{cases} \mathbf{impure} & \text{if } \overline{E} \neq \varnothing \\ \mathbf{pure} & \text{else} \end{cases}$$

We adopted the term *effect class* from Gifford and Lucassen [1986]. In their work on combining functional and imperative programming languages, they look at all possible combinations of reading, writing, and allocation of memory locations by expressions with the goal of classifying these expressions accordingly. Because we do not consider allocation effects in this work (pure procedures are allowed to allocate), the adapted classification scheme of Gifford and Lucassen looks as follows, with the additional **observer** class of procedures that do not generate observable side effects but do observe external side effects.

$$\mathit{Class} = \{\mathbf{procedure}, \mathbf{observer}, \mathbf{pure}\}$$
$$\mathit{classify}(\overline{E}) = \begin{cases} \mathbf{procedure} & \text{if } \mathbf{gen} \in \overline{E} \\ \mathbf{observer} & \text{if } \overline{E} = \{\mathbf{obs}\} \\ \mathbf{pure} & \text{else} \end{cases}$$

6.4 Discussion

Dependence of a procedure on an external side effect (**obs**) is checked by detecting a read–write–read sequence on the same resource: a procedure first reads from a resource that subsequently is modified and read again by

the same procedure. Yet the occurrence of a *single* observable write effect suffices for a procedure to be considered as generating an observable side effect (**gen**).

Requiring only a single observable read effect to determine dependence on external side effects would be too conservative: whenever a procedure applies another procedure for example, it would already be considered as being (potentially) dependent on the applied procedure, although chances are low that procedures—especially primitives—are ever reassigned. Verifying a read–write sequence of a resource would be more precise, but with concrete semantics in mind we opted for a read–write–read sequence to best capture the notion of dependence on an external side effect. A procedure has to observe the external write effect to be dependent on it.

It could be argued that a write–read sequence must be used to detect whether a procedure generates an observable side effect: a procedure first writes to a resource that subsequently is read somewhere. In this case, we decided that the rest of the program does not have to observe the write effect for the procedure to be considered as generating an observable write effect. The fact that the store reflects the modification in a part that is reachable by the rest of the program is sufficient.

6.5 Related work

There exists a large body of work on purity and closely related concepts such as side-effect analysis, referential transparency, and memoization. We compare the most closely related existing work to our approach.

Salcianu and Rinard [Salcianu and Rinard, 2005] present a purity analysis that is based on an underlying pointer analysis (JPPA). Their analysis first constructs parameterized points-to graphs for every method, in which nodes are objects and edges are heap references. A later interprocedural step instantiates these points-to graphs at every call. The goal of the analysis is to distinguish objects allocated during invocation of a method from objects that already exist in the caller state.

Pearce introduces JPure [Pearce, 2011], a modular purity analysis based on annotations that express freshness and locality, two concepts that map closely to how we use these terms in this work. When the focus is on maintaining purity of methods that are annotated as being pure, no whole-program interprocedural analysis is required. JPure is rooted in Java, and can automatically infer annotations. Somewhat similar to our freshness analysis,

JPure starts from an intraprocedural dataflow analysis to model freshness and locality of object references. Purity inference then interprocedurally propagates information using static class hierarchy.

Rytz et al. [Rytz et al., 2013] present a modular type-and-effect system for purity in Scala, which is strongly influenced by JPure. Their system adds support for closures, but annotations cannot be inferred. Their effect system is flow-insensitive to avoid complex event orderings in higher-order functions, while our work is both flow-sensitive and capable of handling higher-order constructs.

Madhavan et al. [Madhavan et al., 2011] present the pointer analysis used in Salcianu and Rinard [Salcianu and Rinard, 2005] as an abstract interpretation. Our work is also based on abstract interpretation, but is different from the original JPPA formulation and thus also from its abstract interpretation reformulation.

Huang et al. [Huang et al., 2012] present ReImInfer, a type inference analysis for reference immutability in Java. ReIm, the underlying type system, qualifies references as being *readonly*, *mutable*, or *polyread*, the latter signifying that a reference is immutable in the current context but may not be in other contexts. Methods are pure if none of their parameters, including `this`, are inferred to be mutable. Like the purity analysis we present in this chapter, ReImInfer is context-sensitive when applying viewpoint adaptation. Viewpoint adaptation adapts the type of properties based on the receiver on field access [Dietl et al., 2007]. Application contexts generated by AAC and P4F are more precise than approaches discussed in [Huang et al., 2012], but at the expense of running time and scalability. ReImInfer does not handle higher-order language constructs.

Pitidis and Sagonas [Pitidis and Sagonas, 2011] treat purity in the setting of a functional language (Erlang), and they take the stricter definition of purity where a function may not depend on external side effects either. Their static analysis tool detects different flavors of purity that map to our classification of functions. Higher-order functions are supported, but the analysis is based on a "pretty simple dataflow analysis", while in this work we employ a state-of-the-art flow analysis that offers maximal call/return precision.

The approach in Finifter et al. [Finifter et al., 2008] is based on static typing. Purity of methods can be enforced by statically declaring all parameters as having an immutable type. Declaring parameters as immutable is sufficient in a language that does not have closures, but would not apply in our setting of a semantics with closures. It is unclear what the impact is of

static restrictions (types, annotations, object capabilities) in systems where pure and impure (or untrusted) code has to be composed. This is potentially impossible or becomes more difficult, or the system has to provide wildcards to make programs pass the static checker.

Naumann [2007] discusses degrees of purity in the context of program specification and verification, which we translate to the procedure purity setting of this chapter. A *strongly pure* procedure has no effects. A *weakly pure* procedure does not mutate caller state but is allowed to allocate objects. *Observational purity* allows modification of the caller state as long as the effects involved are encapsulated "in a suitable sense". In our approach allocation itself does not directly impact purity, so the notion of purity characterized as "does not generates observable side effects" in this chapter corresponds to weak purity in the work of Naumann. The encapsulation that Naumann mentions with respect to observational purity is realized in this dissertation by reasoning about observability of side effects in terms of application contexts.

6.5.1 Advantages Over Existing Approaches

Compared to existing techniques and tools, we believe the purity analysis we presented offers a number of conceptual and practical advantages. Some individual features such as support for closures, using stack reachability when determining extents of side effects, and detection of dependence on external side effects, are rare or absent in existing work. Combining these features results in a purity analysis that, to the best of our knowledge, is unique in its approach. We now discuss these advantages.

Based on an abstract machine approach Our approach is based on an abstract machine that performs an abstract interpretation of a program. We use the AAM technique [Van Horn and Might, 2010], which is a simple method for abstracting these abstract machines so they can be used for static analysis. The same abstract machine can also be parameterized to implement different concrete and abstract semantics, which is useful for correctness and testing purposes [Johnson and Van Horn, 2014]. Abstract machine interpreters are close to regular interpreters, and are therefore well suited to express and instrument the semantics of a language. We believe our approach inherits or at least is positively influenced by these properties of abstract machines and the AAM method.

Our work is the only purity analysis that bases itself on abstract machines and the AAM method. Related work that explicitly mentions abstract interpretation as a foundation is sparse. As we have mentioned in Section 3.6, side-effect and purity analysis is mostly carried out in the setting of type systems.

Support for first-class closures A large part of existing work related to purity targets languages that do not have closures and first-class procedures, such as Java. [Finifter et al., 2008, Huang et al., 2012, Madhavan et al., 2011, Pearce, 2011, Salcianu and Rinard, 2005]. SCHEME$_0$, however, does have first-class closures, meaning that any purity analysis for SCHEME$_0$ must treat variables as proper resources, as is the case in our approach.

Related work that does handle higher-order functional languages includes Rytz et al. [Rytz et al., 2013] (Scala) and Pitidis and Sagonas [Pitidis and Sagonas, 2011] (Erlang). Our approach was designed with explicit support for closures as first-class values.

No dependence on static typing or other annotations Our approach does not require users or module developers to annotate their code, for example by indicating which procedures return fresh objects or by adding static type annotations. Some of the existing approaches such as JPure [Pearce, 2011] require explicit annotations to maintain purity, although the annotations can be inferred. The work of Rytz et al. [Rytz et al., 2013] is strongly influenced by JPure, and therefore also relies on annotations, while our analysis does not.

Determining extent of side effects through stack reachability Traversing the stack to determine the extents of side effects is a novel approach. While some approaches such as JPPA distinguish between the topmost application context and all others when for example masking effects on objects created by a method, our approach potentially examines the entire stack to determine the extent of a side effect. After a straightforward conversion of the simple program in Example 6.1 to Java, JPPA imprecisely reports that all three methods are impure. JPure only explicitly tags method `h` as impure. These results demonstrate that these tools do not perform stack traversal for precisely determining the observability of side effects.

Detection of dependence on external side effects The majority of related work considers a procedure as pure if it does not generate observable side effects. Our purity analysis is able to detect a stricter definition of purity in which a procedure may also not depend on external side effects either. Pitidis and Sagonas [Pitidis and Sagonas, 2011] and Finifter et al. [Finifter et al., 2008] also consider this stricter definition of purity, and the latter accomplishes this by restricting the language and enforcing immutability of the global scope for example. In our analysis we take an optimistic approach by allowing unrestricted access and mutability and detecting when this leads to impurity.

6.5.2 Limitations

An inherent limitation of the purity analysis we presented is that it is a *whole-program analysis* that—given the requirements for determining side effects and their extent and depending on the actual parameterization—can be *costly* to perform. A modular analysis, on the contrary, can be performed on parts of the program. This usually results in a more performant analysis, especially in scenarios where an input program is reanalyzed after only parts of it are modified. At this point it is unknown whether and how the techniques proposed in this work can be modularized.

Other related work also describes tools that require an underlying whole-program analysis [Madhavan et al., 2011, Salcianu and Rinard, 2005], while alternate approaches—primarily those rooted in type systems—are modular in nature [Finifter et al., 2008, Huang et al., 2012, Pearce, 2011, Rytz et al., 2013].

6.6 Conclusion

We presented a purity analysis for SCHEME_0 based on effects reported by general side-effect analysis, and observability determined through stack reachability by procedure side-effect analysis. Purity analysis summarizes the side-effecting behavior of procedures with respect to the rest of the program. For each application of a procedure, we determine whether the application generates observable effects. For dependence on external side effects, we check for a sequence where two applications of the same procedure read the same resource that is modified in between those applications. Based on their

side-effecting behavior, we classify procedures as either **pure**, **observer**, or **procedure**.

Purity analysis is useful for a number of software engineering and optimization purposes. Determining the side-effecting behavior of procedures facilitates reasoning over imperative programs for both humans and tools. As a result, our work can serve as the foundation for further techniques and analyses for program parallelization, function memoization, and detection of referential transparency.

Chapter 7

Implementation

7.1 Introduction

As a proof of concept, we implemented our purity analysis in the Scheme dialect Racket. The resulting tool for determining procedure purity in Scheme programs is called `purity.rkt`, and incorporates all analyses presented in this dissertation. Compared to input language SCHEME_0 for the abstract machine described in Section 2.3, the tool supports a more extensive input language featuring conditionals, vectors, and additional primitives. Any differences between the formalisms presented in earlier chapters and the actual implementation do not fundamentally change the results our analyses produce. The tool's implementation, including benchmark programs and test setup, is publicly available at `https://github.com/jensnicolay/purity/tree/phd`.

We use `purity.rkt` to evaluate our approach by performing a number of experiments described in Chapter 8. In these experiments we make use of two abstract machine configurations: a concrete configuration, and an abstract configuration. The concrete configuration offers full precision, and semantically overlaps with the underlying Scheme runtime. The abstract configuration uses a set lattice in which atomic values are abstracted to their types. The parameters that make up a machine configuration are a lattice, a store update policy, and address allocators for the value and stack stores.

In this chapter, we sketch the general architecture and design of `purity.rkt` (Section 7.2), and how it can be configured. We also discuss the implementation of the two lattices aligning with the configurations used during experimentation: a concrete lattice, and an abstract type lattice (Section 7.3).

We describe how the flow analysis is configured, and describe three optimizations of the abstract machine that improve performance: global stores, stack pop with a metafunction, and atomicity checks (Section 7.4).

The largest deviation from the formalization of the previous chapters constitutes the use of a global value store and global stack store in the abstract machine. If the value store is not updated monotonically, as is the case in concrete semantics, then the value store is a component of the machine states. We give an operational definition of state space exploration in Section 7.4.4.

To decide address freshness, application contexts need to be linked to caller stores (Section 3.4). When using allocation strategies like P4F [Gilray et al., 2016], which produce addresses that do not contain a caller store, a caller-store analysis is required to link application contexts to caller stores. We give our implementation of a caller-store analysis in Section 7.5.

We discuss the implementation of procedure side-effect analysis (Section 7.6), which is parameterized by both observability and escape information, and explain how the implementation of the observability predicate is more generic than and able to simulate its formal counterparts $observable_A$, $observable_F$, and $observable_E$.

We end this chapter by discussing purity analysis, which we implemented in a flow-sensitive manner with per-state widening for the read and observer tables (Section 7.7).

7.2 Architecture and Design

Figure 7.1 depicts the analyses, and the flow of information between analyses that are part of the proof-of-concept implementation. The overall purity analysis comprises the execution of a series of analyses that starts with flow analysis instrumented with effects (Section 3.2). On top of the flow graph produced by the flow analysis, we run escape analysis (Section 5.3), object freshness analysis (Section 4.4.2), procedure side-effect analysis (Section 3.3), and purity analysis (Section 6.2). The results from purity analysis are mapped to effect classes (Section 6.3).

Variable freshness analysis is not implemented as a separate analysis, but has been incorporated in the predicates for observability used to configure the procedure side-effect analysis.

The result of each analysis is captured in a data structure (most often a map), which is consumed by one or more subsequent analyses. Often, the

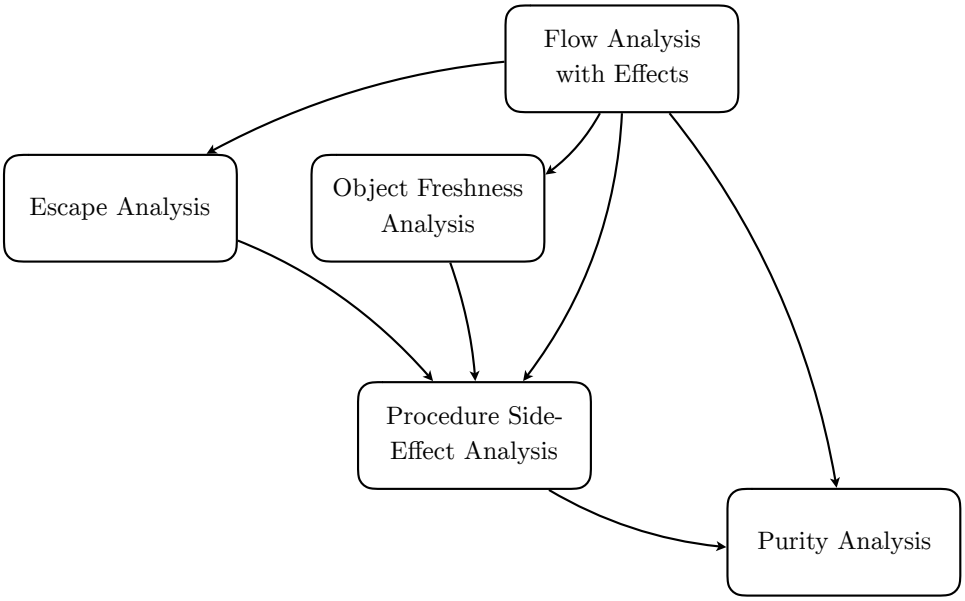


Figure 7.1: Organization of the different analyses that make up purity analyses. Arrows indicate flow of information.

interface between the analyses is a predicate that encapsulates the raw result map. Varying the implementation of these predicates enables the testing of different configurations of the analyses that make up our overall purity analysis. For example, procedure side-effect analysis is parameterized by predicate `escapes?` that answers whether a procedure has been recognized as escaping or not. The result of our escape analysis is a set of potentially escaping procedures, which can be wrapped in a predicate and passed to procedure side-effect analysis. If escape information is unavailable, then the predicate is defined as `(lambda _ #t)`, which conservatively makes all procedures escape.

7.3 Lattices

Structure `lattice` encodes lattices (Appendix B.2) and consists of fixed lattice operations, the bottom value, and a global environment of additional lattice operations.

```
(struct lattice (α γ ⊥ ⊔ ⊑ true? false? eq? global))
```

Lattice operations α, γ, $\bot$, $\sqcup$, and $\sqsubseteq$ operate on lattice values and correspond to the operations discussed in Appendix B.2. At the implementation

level, there are some important subtleties that have to be taken into account by a `lattice` that stem from the fact that some values are created by the lattice (e.g., as the result of abstraction), while others are created outside the lattice—in the abstract machine or one of its parameters in `purity.rkt`. Essentially, we distinguish between three kinds of values.

Atomic values are all values that have an atomic Scheme data type. In `purity.rkt` these are booleans, symbols, characters, numbers, and strings (but not the empty list). A `lattice` operates directly on these values, and is allowed to substitute them with abstractions that lose precision. This means that concretization is possibly undefined for these values.

External values are all other values, and are treated as opaque by a `lattice`. External values are created and manipulated outside the lattice and only "pass through" the lattice. External values are representations of addresses, closures, pairs, vectors, etc. A `lattice` must represent external values with full precision, meaning that abstraction and concretization for these values are each other's inverse. External values often contain abstracted atomic values.

Lattice values are composed of atomic values and external values, and implement the actual lattice operations such as join and subsumption.

Predicates `true?` and `false?` are used to test whether a lattice value represents a true and false (`#f`) value, respectively. These predicates are used for the evaluation of conditionals (e.g., `if`) in the abstract machine. In an abstract semantics, it is possible that an abstract lattice value is both `true?` and `false?`.

Predicate `eq?` encodes equivalence semantics of atomic values only. Equivalence of lattice values is implemented at the level of the abstract machine, which knows how to do this for its own values (i.e., addresses, pairs, etc.), and delegates to `lattice` for all other values.

The lattice operations `true?`, `false?`, and `eq?` in the definition of `lattice` are the minimal set of operations required by the abstract machine to implement Scheme semantics. All other operations on lattice values not provided by the machine are registered in the `global` environment of the lattice, mapping names to Scheme primitives. Examples of primitive operations contained in the global lattice environment are arithmetic operations (`+`, `modulo`, `even?`, ...), atomic type tests (`symbol?`, `char?`, ...), etc., but not `cons`, `vector-set!`, or `eq?` for example.

We have implemented two lattices for experimentation and evaluation: a concrete lattice to support concrete semantics, and a type lattice to support abstract semantics.

7.3.1 Concrete Lattice

The concrete lattice borrows from the underlying Racket language for its atomic values. This is both possible and useful because Racket is a dialect of Scheme.

```
(define conc-lattice
  (lattice conc-α conc-γ conc-⊥ conc-⊔ conc-⊑
           conc-true? conc-false? conc-eq? conc-global))
```

- The abstraction function is the identity function.

  ```
  (define (conc-α d) d)
  ```

- The concretization function turns a lattice value into a singleton set containing that value.

  ```
  (define (conc-γ d) (set d))
  ```

- The bottom element is a value different from all other values.

  ```
  (define conc-⊥ (gensym))
  ```

- Joining two distinct values leads to a precision loss. Because concrete values are always fully precise, join is not defined for concrete values.

  ```
  (define (conc-⊔ d1 d2) (error "illegal operation"))
  ```

- The set of concrete values is a flat order. A concrete value subsumes only the least element conc-⊥ and itself.

  ```
  (define (conc-⊑ d1 d2)
    (or (eq? d1 conc-⊥)
        (eq? d1 d2)))
  ```

- The predicate for recognizing true values semantically overlaps with the underlying Racket semantics, i.e., all values except #f are true.

  ```
  (define (conc-true? d) d)
  ```

- The predicate for detecting the false value also overlaps with the underlying Scheme semantics.

```
(define (conc-false? d) (not d))
```

- Equality of concrete atomic values has Scheme's `eq?` semantics.

```
(define conc-eq? eq?)
```

- The global environment, mapping names to Scheme primitives, directly wraps primitives from underlying Racket in a `prim` structure, which takes a name and a procedure. It is possible to use Scheme primitives directly from Racket because concrete atomic values are Scheme values.

```
(define conc-global
  `(
    ...
    ("+" . ,(conc-α (prim "+" +)))
    ...
   ))
```

7.3.2 Abstract Type Lattice

The abstract lattice implementation encodes atomic values as types, and lattice values as sets.

```
(define type-lattice
  (lattice type-α type-γ type-⊥ type-⊔ type-⊑
           type-true? type-false? type-eq? type-global))
```

- The abstraction function abstract an atomic value into a singleton set representing its type. External values are also converted into a singleton set, but the value itself is not modified.

```
(define (type-α v)
  (cond
    ((number? v) (set NUM))
    ((boolean? v) (set BOOL))
    ((symbol? v) (set SYM))
    ((string? v) (set STR))
    ((char? v) (set CHAR))
    (else (set v))))           ; external value
```

- The concretization function returns the lattice value.

```
(define (type-γ d) d)
```

- The bottom element is the empty set.

```
(define type-⊥ (set))
```

- The join of two lattice values is set union.

```
(define type-⊔ set-union)
```

- Lattice values are ordered by set inclusion.

```
(define type-⊑ subset?)
```

- Every lattice value is considered true, because `type-`α`(#f) = (set BOOL)`, which does not distinguish between non-false values and `#f`.

```
(define (type-true? d) #t)
```

- Every lattice value is also false. This is more imprecise than need be, since lattice values that do not contain `BOOL` can never be an abstraction of a set of concrete values containing `#f`. The usefulness of the more precise encoding is however limited in practice, since an analysis of a non-trivial program eventually visits all branches of a conditional anyway.

```
(define (type-false? d) #t)
```

- The equality test for atomic values returns an abstracted boolean.

```
(define (type-eq? v1 v2) (set BOOL))
```

- The global environment abstracts a Scheme primitive essentially into a function that encodes the return type of that primitive.

```
(define type-global
    `(
      ...
      ("+" . ,(type-α (prim "+" (lambda _ (set NUM)))))
      ...
     ))
```

7.4 Flow Analysis with Side Effects

Now that we have discussed the architecture and lattice configurations of `purity.rkt`, we will look at the implementation of some of the individual

analyses in the remainder of this chapter. We start with flow analysis, the first analysis performed on SCHEME$_0$ programs by our tool.

Flow analysis is performed by the abstract machine (Section 2.4) configured with a lattice, a value store address allocator, a stack store address allocator, and a flag indicating whether value store updates are monotonic (using $\sqcup$) or use assignment.

```
(define (make-machine lattice alloc kalloc mono-store)
(struct system (initial graph state-σ Ξ))
```

The result of flow analysis is a graph, represented as an initial state (`initial`) and a map of transitions from states to successor states annotated with effects (`graph`), and the global stack store (Ξ). Function `state-`σ extracts the value store from a state, abstracting over the location where value stores are kept: inside states, or as a single global store.

We implemented three important optimizations in `purity.rkt` at the level of flow analysis and the abstract machine: the use of global stores and store widening (Section 7.4.1), the avoidance of administrative pop steps (Section 7.4.2), and atomicity checks (Section 7.4.3). We end the section by giving an operational definition of the state-space exploration algorithm (Section 7.4.4).

7.4.1 Global Stores

The following `structs` define evaluation and continuation states in `purity.rkt`.

```
(struct ev (e ρ σ ι κ))
(struct ko (v σ ι κ))
```

All states contain a value store (σ) component, even though the values store is a global component of the abstract machine. The reason is that states need to refer to a specific instance of a value store when the value store is not updated monotonically (Section 7.4.1).

The stack store (Ξ) is not a part of an individual state. Like the value store, the stack store is a global component of the abstract machine, but states never need a reference to a specific instance of a stack store because it grows monotonically (Section 7.4.1).

Value Store

We implemented flow analysis with a global value store. We opted for a global store because this choice has no impact on the results of the concrete

semantics, and with abstract semantics maintaining a separate store per state takes the machine longer to reach a fixpoint if it exists [Gilray et al., 2016].

In concrete semantics, a single global value store poses no additional difficulties. The sets of states to be explored (the "frontier") always consists of a single state, because the machine only explores a single path through the code at a time (no transition rule yields more than one successor state). Therefore, the global store *is* the store of the single frontier state. However, concrete semantics uses strong updating when modifying the store (Section 2.4.3), replacing existing values by new values, which can result in non-monotonic updates. As a consequence, visited states in the seen set must be paired with the global store that was in effect at the time of their exploration.

With abstract semantics, a global value store is used to implement store widening [Van Horn and Might, 2010]. The global store is shared between all states, and only updated monotonically during state exploration by joining existing values with new values. At any time during exploration it subsumes the stores of all the states explored so far. Because the final store then subsumes all intermediary stores, the store can be omitted in the states. As with the stack store, the set of visited states must be cleared when the global value store increases to guarantee that every state is visited with its largest attainable store.

Flag `mono-store` controls the behavior of the value store only (the stack store is global and updated monotonically by default). If the flag is true, then individual states do not contain a store (σ inside states is set to `#f`), and during exploration the set of visited states is cleared when the global store increases. Requesting the store for a state using `state-`σ returns the global store. When `mono-store` is false, then the global store at the time of exploration becomes the store of the visited state, and function `state-`σ returns this store.

Stack Store

The stack store is also a global component of the abstract machine in `purity.rkt`. Because the stack store represents control flow, which is monotonic in nature (the machine can only discover *additional* paths through a program). Contrary to the state-space given in Figures 2.3 and 3.1 on page 19 and on page 41, the stack store is not implemented as a component of a state. Instead, whenever the stack store increases, the set of seen states

is cleared so that every state eventually will be visited with the largest attainable stack store for that state.

7.4.2 Pop Metafunction

The abstract machine from Section 2.4, and its instrumented version in Section 3.2, has a transition rule [K-RET] for continuation states with an empty local continuation. The rule looks up underlying stacks by dereferencing the state's stack address in the stack store, and produces a successor continuation state per underlying stack. As suggested in Johnson and Van Horn [2014], it is more optimal to delegate to a metafunction for popping the stack instead of using an "administrative" pop step [K-RET] on the level of the machine.

Function *pop* from Section 4.4.2 is the metafunction we are looking for, and in `purity.rkt` the transition rule [K-RET] delegates to *pop* instead of doing a single stack address dereference. The metafunction pops the stack until the local continuation is not empty, or the bottom of the stack is reached.

For tail calls, using an administrative machine step for popping the stack produces more states than using metafunction *pop*. These extra states have to be considered in the fixpoint computation, which then needs to perform more expensive "seen" checks than *pop*—the latter only needing to keep track of a generally small set of seen contexts [Johnson and Van Horn, 2014].

Example 7.1. In Figure 2.4b on page 28, the machine has computed the result value of the program in ς_{13}. It then takes three consecutive administrative pops to reach the program exit in ς_{16}, reflecting the fact that all applications on the call stack are in tail position when the program ends. Using metafunction *pop* in transition rule [K-RET], the machine would transition directly from ς_{13} to final state ς_{16}.

7.4.3 Atomicity Checks

Input language SCHEME_0 (Figure 9.1 on page 171) allows any kind of expression in certain subexpression positions, for example in the binding expression of a `let`. Transition rule [E-LET] of the abstract machine evaluates the binding expression, while pushing a continuation to bind the resulting value and continue with the evaluation of the `let` body. This requires at least an evaluation state and a continuation state for the binding expression and its value. However, if the binding expression is atomic, it can be evaluated

using the atomic evaluator, in which case no continuation needs to be pushed (Section 2.3.1). This is computationally less expensive, and moreover the benefit is accumulative because fixpoint computation generally takes more time with more states in the graph.

However, there is an important downside to checking for atomicity of an expression to avoid more complex non-atomic evaluation. It forces clients of the flow analysis to also check whether certain syntax positions are atomic expressions, rendering this optimization "leaky". As the results of atomic evaluation are not directly reflected in the graph, clients need to invoke the atomic evaluator to reconstruct them.

We chose a pragmatic solution in `purity.rkt`, balancing performance gains against the disadvantages for client analyses that operate on flow graphs, of which there are four in our approach. Due to the prevalence of atomic binding expressions in `let` and especially `letrec`—the latter used almost exclusively to bind recursive procedures—the abstract machine checks the atomicity of the binding expression in only those constructs.

Example 7.2. In Figure 2.4b on page 28, the machine needs 16 steps to compute the result. Using metafunction *pop* instead of an administrative pop step (Example 7.1) reduces the number of steps to 14. Adding atomicity checks as described in this section further reduces the number of steps to 8. Evaluation and continuation states binding `lambda`s to variables are no longer in the graph, but instead are performed atomically during the [E-LET] transition. This means states ς_1, ς_2, ς_4, ς_5, ς_8, and ς_9 are no longer present in the graph.

7.4.4 State Exploration

Unlike Section 2.5, which intensionally defined state exploration as a reflexive transitive closure of the transition relation, this section takes an *operational* look at state exploration.

Figure 7.2 contains the implementation of procedure `explore`, which performs the main state exploration loop. State exploration keeps a set of seen states `S`, a set of states to explore `W`, and the graph that accumulates edges annotated with effects `graph`. If the work list is empty, then the flow graph is returned (line 3). Otherwise, the algorithm pulls a state from the work list (line 4). If it is a state that was previously encountered, exploration continues with the seen state removed from the work list (line 5). Lines 6 and 7 save the current value of the global timestamps σ`i` and Ξ`i` for value

```
1  (define (explore S W graph)
2   (if (set-empty? W)
3       graph
4       (let* ((s (set-first W)))
5        (if (set-member? S s)
6            (explore S (set-rest W) graph)
7            (let* ((old-σi σi)
8                   (old-Ξi Ξi))
9              (let-values (((W* graph*)
10                           (handle-transitions (set-rest W)
11                                               graph
12                                               s)))
13               (let ((S* (if (or (> Ξi old-Ξi)
14                                 (and mono-store (> σi old-σi)))
15                             (set)
16                             (set-add S s))))
17                (explore S* W* graph*))))))))
```

Figure 7.2: Implementation of the state exploration algorithm with global stores and timestamps.

store and stack store, respectively. In case the store grows monotonically, the timestamp represents the number of times the store has increased. This allows for efficient comparison of stores by checking equality of integers. On line 10 the algorithm delegates to `handle-transitions`, passing in the work list with the currently visited state removed, the current graph, and the visited state. This results in an updated work list `W*` and graph `graph*`. On lines 13 and 14 the current timestamps are compared with the saved timestamps. If they have increased, the set of seen states `S` is cleared; otherwise the visited state is added to `S`. The timestamp check for the value store only happens if that store grows monotonically, which is denoted by flag `mono-store`. On the last line of the algorithm, state exploration continues with the updated components. State exploration starts with an empty set of seen states `S`, a set of states to process `W` containing the initial state obtained after injecting the program, and an empty graph `graph`.

```
(explore (set) (set initial) (hash))
```

When the abstract machine steps a state using procedure `step`, this returns a set of transitions. A transition contains a successor state and a set of effects that occur on transition to the successor.

```
(struct transition (s E))
```

Procedure `handle-transitions` steps a state and iterates over every outgoing transition (line 2). For each transition, it adds the successor state `s*` to the work list (line 5). It also adds the set of effects `E` of the transition to the (possibly empty) set of effects that was already seen for the transition (lines 7–9).

```
1 (define (handle-transitions W graph s)
2   (for/fold ((W W) (graph graph)) ((t (step s)))
3     (match t
4       ((transition s* E)
5        (let ((W* (set-add W s*))
6              (graph*
7               (let* ((hs (hash-ref graph s (hash)))
8                      (E* (set-union E (hash-ref hs s* (set)))))
9                 (hash-set graph s (hash-set hs s* E*)))))
10          (values W* graph*))))))
```

`purity.rkt` uses two representations for graphs. During state exploration, graphs are represented as a map from source states to another map from successor states to effects. This representation is computationally less expensive to update and maintain during analysis. After state exploration, this graph is transformed into a map from source states to a set transitions consisting of successor states and effects.

7.5 Caller-Store Analysis

Caller-store analysis is required if the application context does not contain caller stores (Section 3.4). The stack address allocator in the original AAC approach [Johnson and Van Horn, 2014] generates application contexts that do include the caller store, but the more recent P4F approach [Gilray et al., 2016] does not. Since we used the P4F allocator for our experiments in Chapter 8, we therefore run caller-store analysis prior to procedure side-effect analysis.

Caller-store analysis takes a system `sys` (the result of flow analysis, see Section 7.4), and returns a map `ctx->addr` from context to the set of addresses reachable in the caller store for that context. Lines 2–5 obtain the necessary functions and data from the system, needed by the analysis. The analysis itself is defined as an iteration over all states in the flow graph (line 6), and a nested iteration over the outgoing transitions of a state (line 9). It looks for an application evaluation state (line 8), followed by a body evaluation state (line 11). Lines 12–15 compute the set of reachable addresses

in the caller store, and add it to the set of already reachable addresses `A` for the entered application context $\hat{\kappa}\star$. Procedures `reachable` and `referenced` correspond to functions $\mathcal{R}$ and $\mathcal{T}$, respectively (Section 3.5). Concretization function γ is required by $\mathcal{R}$ to obtain a set of non-atomic concrete values from a lattice value in the store that need to be traced.

```
(define (caller-store-analysis sys)
 (define graph (system-graph sys))
 (define state-σ (system-state-σ sys))
 (define Ξ (system-Ξ sys))
 (define γ (lattice-γ (system-lattice sys)))
 (for/fold ((ctx->addrs (hash))) (((s ts) (in-hash graph)))
   (match s
     ((ev (? «app»? e) ρ σi ι κ)
      (for/fold ((ctx->addrs ctx->addrs)) ((t (in-set ts)))
        (match t
          ((transition (ev _ _ _ '() κ*) _)
           (let* ((A (hash-ref ctx->addrs κ* (set)))
                  (σ (state-σ σi))
                  (A* (set-union A (reachable (referenced s Ξ)
                                              σ γ))))
             (hash-set ctx->addrs κ* A*)))
          (_ ctx->addrs))))
     (_ ctx->addrs))))
```

7.6 Procedure Side-Effect Analysis

Observability

The implementation contains a parameterized observability predicate `observable?`. By providing appropriate implementations of its three freshness predicate parameters, it can be configured to behave like any of the three central observability predicates described in this dissertation: $observable_A$ from Section 3.4, $observable_F$ from Section 4.5.2, and $observable_E$ from Section 5.4.2.

```
(define (observable? fresh-addr? fresh-var? fresh-obj?)
  (lambda (eff κ ς stat)
    (match eff
      ((wv a v)
       (not (or (and stat (fresh-var? v κ))
                (fresh-addr? a κ))))
      ((wf a _ v)
       (not (or (and stat (fresh-obj? v ς κ))
```

```
           (fresh-addr? a κ))))
((rv a v)
 (not (or (and stat (fresh-var? v κ))
           (fresh-addr? a κ))))
((rf a _ v)
 (not (or (and stat (fresh-obj? v ς κ))
           (fresh-addr? a κ)))))))
```

The structures `wv`, `wf`, `rv`, and `rf` represent variable and field reads and writes, respectively. Any type of freshness can be disabled by passing `(lambda _ #f)` as its predicate. Unlike in the formalization, the implementation allows variable freshness and object freshness to be enabled or disabled separately.

The flag `stat` indicates whether the dynamic scope still agrees with the static scope for an application context, and therefore whether variable or object freshness can be used. The flag simulates the top flag (Section 4.5.1) if its value is true only for the topmost application context. The flag can also behave as the escape flag from Section 5.4 when it is updated using the results from escape analysis. Updating `stat` is the responsibility of procedure side-effect analysis, which actually performs stack traversal.

Procedure Side-effect Analysis

Procedure side-effect analysis takes a system and two predicates `observable?` and `escapes?` for determining effect observability and escaping of procedures. It checks the observability of every effect in every application context. A side effect in the context of an application is observable if it is on a resource that is not fresh in that context. If the effect is not observable outside a particular application context, then it is masked in that context. Predicate `escape?` is used as additional contextual information for predicate `observable?`. It can be implemented as a thin wrapper around the result of escape analysis, which returns a set of escaping procedures. In principle, the logic of `escape?` can be folded into `observable?` (in that case passing the traversed portion of the stack to `observable?`), but for simplicity we kept the two predicates separated.

```
(define (side-effect-analysis sys observable? escapes?))
```

The analysis returns a map from states to application contexts to a set of observable side effects. As mentioned before, the term “observable” is used for both read and write effects.

7.7 Purity Analysis

Purity analysis takes a flow graph as input in order to examine all effects that are produced during program execution. This flow graph is combined with effect observability computed by the procedure side-effect analysis in order to determine whether procedures generate observable side effects or depend on external side effects. The result of purity analysis is a map from procedures to a summary of their side-effecting behavior.

The purity analysis in Section 6.2 uses a global read table and observer table, thereby implicitly implementing a global widening strategy for these tables. This has a negative impact on precision for observers: reads and writes in completely unrelated branches of the program can mark procedures as (potential) observers. We therefore chose to implement purity analysis in a flow-sensitive manner: instead of containing states, the work list and set of seen states contain triples consisting of a state, a read table, and an observer table. Although flow-sensitivity is more precise, it is more expensive to compute. Therefore we added per-state widening of these tables to accelerate fixpoint computation. When pulling a state table, read table, and observer table from the work list, the tables are joined with the previous tables seen for that state, before effects are handled. Because widening happens on a per-state basis, flow-sensitivity is maintained, and we did not observe any loss in precision due to widening.

7.8 Conclusion

This chapter presented an overview of our main technical artifact, `purity.rkt`, which is a purity analysis for Scheme implemented in Racket. We discussed the extensions that `purity.rkt` makes over SCHEME_0 used in the formalization in preceding chapters. We also mentioned important performance and precision considerations that make the implementation more efficient and precise. The main optimizations in `purity.rkt` are the use of global stores and store widening, the avoidance of administrative pop steps, and atomicity checks. We also described a flow-sensitive purity analysis with widening of its tables.

In the next chapter we use `purity.rkt` to evaluate the analyses presented in this dissertation. This chapter introduced an overview, and some configuration and implementation details of these analyses required to understand the experiments we performed.

Chapter 8

Evaluation

8.1 Introduction

In this chapter, we report on several experiments we ran to evaluate correctness, soundness, performance, and precision of the techniques and analyses presented in this dissertation. Experiments were run on a machine equipped with a quad-core 2.8 GHz Intel Core i7 processor, and 16 GB of DDR3 memory. We used Racket v6.4 [Flatt, 2010] as the runtime. Although there were 4 physical (8 logical) cores available on the test machine, our implementation is single-threaded and does not benefit from additional cores.

All results presented in this chapter are produced by a test runner that analyzes each program in a set of benchmarks and computes the test results. The analysis phase consist of the execution of a series of analyses described in this dissertation (Figure 7.1), starting with flow analysis and ending with purity classification. This series of analysis is run using different configurations (concrete vs. abstract) and combinations of the analyses involved (with or without freshness, etc.). In addition to the direct analysis result, analyses also compute basic statistics such as their duration. More complex processing is done by the test runner in between and after analyses have completed.

We did not measure memory consumption in detail, but it was possible to execute the test runner for the complete set of benchmarks when the available memory was limited to 4 GB. Increasing the limit to more than 4 GB did not significantly impact timings. Limiting available memory to 2 GB still allowed the test setup to finish, but significantly increased the running time of our test runner due to garbage collection overhead.

We start by defining the terminology we use when evaluating our approach (Section 8.1.2). We also give an overview of the test configurations used by the test runner (Section 8.1.1), and discuss the benchmark program and the general test setup (Section 8.1.3). The remainder of the chapter is devoted to describing the experiments we performed with the different analyses presented in this dissertation, and a discussion of the obtained results. We end this chapter with a number of general conclusions supported by the outcome of the experiments.

8.1.1 Configurations

The abstract machine from Chapter 3 is parameterized by a lattice, value and stack store allocators, a store update policy, and a flag indicating whether the store is updated monotonically. This enables configuring the machine to express concrete and different abstract semantics for SCHEME_0 without having to modify the abstract machine or any of the analyses.

For the purpose of our experiments, we define two configurations: a concrete configuration and an abstract configuration. Most of the details of these configurations were given in Chapter 7, but we summarize them here.

Concrete configuration (CONC) In the concrete configuration, the abstract machine is configured with a concrete address allocator that uses an integer counter, which is incremented with each allocation. Stack addresses are generated using the P4F allocation [Gilray et al., 2016] strategy extended with a counter to generate unique addresses. The concrete configuration uses a concrete lattice for values and operations, which borrows from the underlying Racket runtime (Section 7.3.1). The store is updated using strong updating instead of joining. The value store is kept inside the machine states, while the stack store is global.

Abstract configuration (ABST) The abstract configuration uses the 0CFA abstract address allocator [Shivers, 1991] that uses syntactic elements as addresses. The AST node representing the declaration of a variable is that variable's address, and the node responsible for creating an object is the created object's address. Stack addresses are generated using the P4F approach, so they contain the applied procedure and the extended environment. The abstract configuration uses the type lattice described in Section 7.3.2. The store is updated using weak updating through join, as presented in the formalization

of the flow analysis in Chapter 3. Both the value and stack store are global, and are not represented as state components.

8.1.2 Terminology

We evaluate our approach by examining the following properties of analyses results.

Correctness When testing correctness, we check that the actual outcome of an experiment is equal to the expected outcome. The expected outcome is specified manually. We test correctness when running our analyses under the concrete configuration, because this configuration most closely reflects actual program semantics and provides the most precision.

Soundness The result of an analysis run under the abstract configuration is sound when it reflects all possibilities that are computed by that same analysis under the concrete configuration. We test soundness by examining the concrete result after abstraction, and then check whether there is a corresponding abstract result that subsumes (i.e., is as precise or less precise) the abstracted concrete result (Appendix B.1).

Precision Precision is tested using a similar setup as used for soundness. A sound abstract analysis is allowed to overapproximate, i.e., it is allowed to reflect possibilities that may never occur under concrete semantics. We quantify overall analysis precision by measuring the amount of *false positives*, which are abstract results that have no concrete counterpart. The fewer false positives, the higher the precision of the abstract analysis. Precision can also be measured in a relative way, between abstract results of different static analyses that compute the same information.

Performance The performance of an analysis is measured as the time it takes for an analysis to compute an answer, and the memory it takes to do so. We test the performance of our implementation to evaluate how it performs on current off-the-shelf hardware, which is a machine equipped with a quad-core 2.8 GHz Intel Core i7 processor, and 16 GB of DDR3 memory. We already discussed memory requirements in the introduction, and we will evaluate the running times of the analyses. For running times, we use ε to indicate a duration of less than a second.

8.1.3 Benchmark Programs

Type of Benchmark Programs

We distinguish between two types of benchmarks.

Unit tests Unit tests are short programs that test the various challenges our analyses have to deal with. They are typically focused on testing one single aspect or feature of the language, such as free variable access, different procedure calling sequences, a procedure being passed as argument, etc. The defining characteristic of these programs is that they are designed in such a way that it is possible to manually specify the expected outcome of an analysis. A unit test for an analysis therefore consists of a program paired with its expected outcome.

Another important characteristic is that coverage of the flow analysis in the concrete configuration is near or at 100%. Such a high coverage implies that the abstract machine visits all or nearly all expressions in the program. There is therefore little to no unreachable code in the program. High coverage is important when checking soundness and precision of abstract results by comparing them with concrete results. Because abstract results should overapproximate the concrete results, higher coverage under the concrete configuration increases confidence in soundness and precision of the approach for each benchmark, and therefore of the overall approach.

Most of the unit tests are hand-crafted programs, although some existing smaller and well-known benchmark programs such as `fib` and `nqueens` are also in the set of unit tests. There are over 80 unit tests in our collection of test programs.

Larger, existing benchmark programs We also included larger, existing benchmark and real-world programs in our set of test programs. In general, these benchmarks are too large to verify manually for correctness, or the confidence in doing so would be low. We performed limited manual inspection during the implementation and evaluation of the analyses for these larger programs.

Existing benchmarks come from a variety of sources: the Gabriel performance benchmarks [Gabriel, 1985], PDCFA benchmarks typically used to challenge control-flow analyses (e.g., [Earl et al., 2012]), JOlden performance benchmarks for Java which we converted to Scheme [jol], and benchmarks from the Computer Language Benchmarks Game

(CLBG) [Fulgham and Gouy, 2009]. There are also some benchmark programs that are not considered to be part of any well-known suite.

The test runner was executed on *all* benchmark programs (unit tests and larger programs) every time the implementation changed. During each test run, correctness is mechanically verified for unit tests which have a manually specified expected outcome. Additionally, soundness of abstract results with respect to concrete results is mechanically verified for all benchmarks by checking that the abstract results subsume the concrete results.

The test runner was also used to automatically produce the results from experiments we report on in the upcoming sections. We report results for a selected set of 13 benchmark programs, consisting of 3 hand-crafted and 10 existing benchmark programs. We chose programs that contain non-trivial side-effecting behavior, or, in the case of `fib` and `nqueens`, programs that are free from side effects. The hand-crafted programs give an idea of how our techniques perform on the edge cases that are typically not present in real-world benchmarks. The larger and existing benchmarks evaluate how well our techniques perform on real-world programs.

Benchmark Programs

Table 8.1 lists the set of 13 programs on which we evaluate our analyses in the remainder of this chapter, together with their size, and the number of times a special form or expression appears in the program that is relevant in terms of side effects. We give a short description of these benchmarks.

1. `fib` is the smallest program in our set and implements a tree-recursive Fibonacci function. This function is an often used example of a procedure that benefits from memoization, because it is pure and performs a lot of redundant computation.

2. `purity65` is a hand-crafted program that implements a recursive procedure that modifies a locally constructed pair bound using a `let` with another `let` as value expression.

3. `purity73` is a hand-crafted program containing two nested procedures that implement loops, with the outer procedure calling the inner and the inner calling both inner and outer procedures. The inner procedure mutates its local variables and also variables from the outer procedure.

4. `purity75` is a hand-crafted program containing two procedures representing loops, in which one procedure takes the other procedure as argument. Both procedures mutate locally constructed pairs.

5. `treeadd` is a program from the JOlden benchmark suite [jol] that we ported from Java to Scheme and extended. It implements three different variations on constructing a binary tree, inspired by the static constructor method and constructors found in the original Java benchmark. After construction, all integer node values of the binary tree are summed through a tree traversal.

6. `nqueens` is a combinatorial problem in which n queens need to be placed on an $n \times n$ chessboard such that no two queens attack each other. The implemented algorithm uses backtracking and no destructive operations.

7. `dderiv` is a Gabriel benchmark program [Gabriel, 1985] that implements table-driven symbolic derivation. Tables are represented using association lists.

8. `destruc` is a Gabriel benchmark program [Gabriel, 1985] that features many destructive list operations.

9. `grid` is a program that implements a multidimensional array package based on vectors [Norskog, 1993].

10. `matrix` is a CLBG benchmark program [Fulgham and Gouy, 2009] that multiplies matrices represented as vectors. We based ourselves on the implementation for Chicken Scheme [chi].

11. `fannkuch` is a benchmark program that performs permutations and reversing of vectors [Anderson and Rettig, 1994]. We again based ourselves on the implementation for Chicken Scheme.

12. `mceval` is a PDCFA benchmark program [Earl et al., 2012] that implements a Scheme interpreter evaluating a tree-recursive Fibonacci.

13. `scm2java` is a PDCFA benchmark program [Earl et al., 2012] that implements a Scheme to Java compiler. The input language of the compiler is close to the input language SCHEME_0 supported by our abstract machine.

8.2 Flow Analysis With Effects

All analyses presented in this dissertation, except for variable freshness analysis and flow analysis itself, rely on the results of flow analysis instrumented with effects from Section 3.2. This flow analysis computes a flow graph of the program, which then can be queried to obtain the program properties of interest for determining procedure purity.

Our flow analysis is a variation on the flow analysis by Johnson and Van Horn [2014], which itself is the latest analysis in a line of analyses for higher-order programs starting with Shivers [1988] (see related work in Section 2.6 for more details). Soundness of these analyses has been proved in the publications that introduce each of these analyses. However, we also perform experiments to evaluate the correctness and soundness of our particular flow analysis implementation.

8.2.1 Correctness

To evaluate correctness of the implementation of our flow analysis, we ran each benchmark program under the concrete configuration, and checked that the abstract machine produced a single answer that is equal to the answer computed by Racket for the same benchmark program. When configured with a concrete value lattice, atomic values used by the abstract machine semantically overlap with the underlying Racket runtime. Other, non-atomic values such as lists cannot be compared directly. For these values we compare their string representations instead, or query certain fields of the resulting compound values.

From this evaluation, we conclude that our implementation is correct with respect to the programs used in our benchmarks.

8.2.2 Soundness

For each program, we checked that the abstract result value subsumes the concrete result value after abstraction (Appendix B).

Result values are obtained from end states in the flow graph, which are continuation states (**ko**) with an empty stack (i.e., $\langle\rangle$ as local and ϵ as meta-continuation). While a finite benchmark produces a linear flow graph with a single result state under concrete configuration, in the abstract it may be necessary to join multiple result states to compute a single abstract result value for the entire program.

Benchmark	Size	lam	set!	cons	set-car!	set-cdr!	make-v	v-set!
fib	42	1	0	0	0	0	0	0
purity65	29	1	0	1	1	0	0	0
purity73	60	2	2	0	0	0	0	0
purity75	80	2	0	2	1	1	0	0
treeadd	392	8	6	5	2	2	0	0
nqueens	247	6	0	4	0	0	0	0
dderiv	616	20	1	29	0	2	0	0
destruc	420	11	0	3	4	3	0	0
grid	382	5	3	3	0	0	1	2
matrix	385	11	3	3	0	0	4	4
fannkuch	465	16	3	0	0	0	2	7
mceval	1687	87	0	28	3	1	0	0
scm2java	1972	64	6	66	0	0	0	0

Table 8.1: Size of 13 selected benchmark programs and the number of times a certain type of expression or special form appears in their program text. *Size* is the number of AST nodes in the program. `lam` stands for a `lambda` expression, `make-v` for `make-vector`, and `v-set!` for `vector-set!`.

	— ABST —			
Benchmark	States	Edges	Coverage	Time
fib	20	30	100.0%	ε
purity65	20	27	100.0%	ε
purity73	29	34	100.0%	ε
purity75	42	52	100.0%	ε
treeadd	216	310	100.0%	ε
nqueens	133	189	100.0%	ε
dderiv	356	1460	90.42%	ε
destruc	238	337	100.0%	ε
grid	237	424	93.46%	ε
matrix	188	239	97.92%	ε
fannkuch	215	350	100.0%	ε
mceval	1224	12103	97.45%	27”
scm2java	1384	13981	95.79%	30”

Table 8.2: Abstract flow analysis results. *States* is the number of states in the computed flow graph, *Egdes* the number of edges. *Coverage* is the percentage of AST nodes visited by the abstract machine interpreter. *Time* is the running time of the flow analysis.

Example 8.1. Running the `fib` program under the CONC configuration produces a single result state with integer `21` as result value. Running the same program under the abstract configuration using a type lattice, produces $\{Num\}$ as a result. The abstract result is sound with respect to the concrete one, because $\{Num\} \sqsupseteq \alpha(21)$. □

8.2.3 Precision

We use a state-of-the-art flow analysis with pushdown precision [Gilray et al., 2016, Johnson and Van Horn, 2014], so we are confident that precision in terms of control flow and value flow is on par with existing flow analyses.

We did not specifically experiment with the precision of the flow analysis itself. The precision of the flow analysis influences the precision of any client analysis built on top of that flow analysis, in our case procedure side-effect analysis, object freshness analysis, escape analysis, and purity analysis. We therefore measure the precision of the flow analysis indirectly through the client analyses, and we do so in the different experiments that follow. These later experiments show that the flow analysis is precise enough to produce results that can be used by client analyses.

We can however give a rough estimate of the precision of flow analysis. Table 8.3 lists the number of states and edges in the flow graph computed by the concrete flow analysis. When the number of states is n, then the number of edges is invariably $n - 1$, which means that the flow graph is linear (in a connected graph, which is the case). Table 8.2 is the abstract counterpart of the previous table. Here, the previous relation between the number of states and edges breaks down, with the flow graph containing more edges than states in all cases. This means that states have multiple successors instead of maximally one as in the concrete case, indicating a loss of precision in the abstract control flow. As a rough measure of control flow precision, we can state that the more the edges outnumber the states, the less control flow precision is obtained. Because control flow and value flow are strongly intertwined, this results in an overall loss of precision.

Generally speaking, abstract control flow is computed relatively precisely. The analysis exhibits the least precision on benchmark programs `mceval` and `scm2java`, with roughly 10 times as many edges as there are states. Inspecting these programs and their computed flow graph indicates that the loss of precision is due to a combination of heavy pair allocation (through `cons` and quoted lists) and heavy branching (through conditionals) during program execution. Pairs are allocated based on their allocation site, and

the abstract type lattice does not distinguish between true and false values. Because lower precision in handling pairs incurs a precision loss in control flow and vice versa, the overall precision is low in the mentioned programs. In the following experiments, we will see that client analyses do not recover from this loss of precision, especially in the case of `mceval`.

8.2.4 Performance

We measured the time it takes for flow analysis to complete, and how many states and edges the resulting flow graph is comprised of. We are primarily interested in the results for the abstract configuration. This is the configuration in which the analysis is meant to be used, as it generates a flow graph that represents a finite overapproximation of every possible concrete program execution.

Table 8.2 shows the number of states and edges in the flow graph computed by the abstract flow analysis, and how long it takes to compute the graph. If we assume that subsecond running times are required for scenarios in which on-the-fly flow information is required, for example as part of a developer feedback mechanism in an IDE, then we conclude that the flow analysis, and therefore any client analysis that consumes a flow graph, is too slow to be used in such a scenario. However, the size of the resulting graph and the time it takes to compute it are reasonable when the graph is to be used as the starting point of an on-demand analysis pipeline.

The flow analysis is a whole-program analysis, and our implementation is expensive to run on large programs. However, abstract flow analysis analyzes all benchmark programs, except `mceval` and `scm2java`, in less than a second. We therefore conjecture that modularizing the flow analysis and making it incremental would enable it to be fast enough to be used in a scenario in which on-the-fly feedback is required. We consider modularizing and incrementalizing the analysis, to the extent that this is possible, as future work.

Because an evaluation of correctness and soundness depends on concrete results, Table 8.3 shows the number of states and edges computed by the concrete flow analysis, together with the coverage and duration. In general, coverage is high, except for `dderiv` which only has 63.15% coverage in the concrete. The reason is that the expression that is derived in the benchmark program does not contain a quotient, while the program supports this (incorrectly—according to an accompanying comment in the source code).

8.3 Escape Analysis

The goal of escape analysis (Chapter 5) is to determine which procedures potentially escape their defining lexical scope. Our procedure side-effect analysis can make use of this knowledge to increase its precision. If, during stack traversal, it is the case that no previously encountered application context is associated with an escaping procedure, scope-based variable and object freshness can be used (Section 5.4).

Table 8.4 lists some statistics concerning procedure application and escaping procedures for the concrete and abstract configurations. In general, comparing the number of called procedures (*Called*) to the number of syntactic procedures (*Procs*) supports our earlier observation that code coverage in our set of benchmarks is high under both configurations, except for `dderiv` (see also Tables 8.2 and 8.3).

Column *Escaping* of Table 8.4 represents the number of procedures reported as potentially escaping by the analysis. Out of the 13 benchmarks, only 5 benchmarks feature escaping procedures, of which only 4 stem from existing (non-synthetic) benchmark suites. Although Scheme is a higher-order language with first-class closures, we found that many existing Scheme benchmarks actually do not feature escaping procedures.

8.3.1 Correctness

For the smaller benchmark programs and unit tests, we manually classified each procedure as escaping or non-escaping, and checked that the escape analysis produced the expected outcome. We found that this is always the case. The escape analysis from Section 5.3 is a straightforward analysis that checks the presence of reachable closure values at program points where they may escape, without requiring a fixpoint computation. These two points make us confident that our escape analysis is correct.

8.3.2 Soundness

We mechanically checked that the set of called procedures under the abstract configuration always is a superset of the set of called procedures in the concrete. Likewise, we verified that the set of escaping procedures under the abstract configuration always is a superset of the set of escaping procedures in the concrete. We found no procedures that were marked as potentially escaping by the abstract analysis, but not by the concrete analysis.

	— CONC —			
Benchmark	States	Edges	Coverage	Time
fib	735	734	100.0%	ε
purity65	20	19	100.0%	ε
purity73	253	252	100.0%	ε
purity75	111	110	100.0%	ε
treeadd	2571	2570	100.0%	ε
nqueens	2704	2703	100.0%	ε
dderiv	2871	2870	63.15%	ε
destruc	2292	2291	95.95%	ε
grid	2019	2018	89.53%	ε
matrix	9401	9400	97.92%	ε
fannkuch	8969	8968	99.78%	ε
mceval	9223	9222	83.17%	ε
scm2java	10610	10609	91.43%	ε

Table 8.3: Concrete flow analysis results. Concrete counterpart of Table 8.2.

		— CONC —		— ABST —	
Benchmark	Procs	Called	Escaping	Called	Escaping
fib	1	1	0	1	0
purity65	1	1	0	1	0
purity73	2	2	0	2	0
purity75	2	2	1	2	1
treeadd	8	8	0	8	0
nqueens	6	6	0	6	0
dderiv	20	10	6	14	6
destruc	11	11	0	11	0
grid	5	5	0	5	0
matrix	11	10	0	10	0
fannkuch	16	16	1	16	1
mceval	87	72	2	85	2
scm2java	64	60	8	61	8

Table 8.4: Called versus total number of procedures, and results from escape analysis for the concrete and abstract configurations. *Procs* is the number of procedures in the benchmark program. *Called* is the number of actually applied procedures during interpretation. *Escaping* is the number of potentially escaping procedures as determined by escape analysis.

8.3.3 Precision

Our escape analysis only detects *potentially* escaping procedures, and also for reasons outlined in Section 8.3.1 our escape analysis is very precise in this regard and in all configurations.

Moreover, in smaller benchmark programs, as well as in programs in the set of 13 benchmark programs on which we report, we found that procedures that are reported as potentially escaping, actually do escape, i.e., are applied outside of their defining environment.

8.3.4 Performance

Relative to the overall running time (Table 8.14), and to the running times of the other individual analyses, escape analysis finishes fast. No benchmark took more than one second to analyze.

8.4 Scope-Based Freshness Analysis

Chapter 4 introduced variable and object freshness analysis based on scopes. The goal of scope-based freshness analysis is to improve the precision of address-based procedure side-effect analysis. Under the abstract configuration, address-based side-effect analysis loses precision because addresses in this configuration are chosen from a limited set to guarantee termination of a static analysis. In Sections 8.5 and 8.6 we carry out experiments to measure how much precision scope-based freshness can recover for abstract procedure side-effect and purity analysis.

Based on scopes of variables, variable freshness is less challenging than object freshness analysis, which tracks object freshness through variables, assignments, and bindings. We therefore focus on testing object freshness analysis in this section.

Scope-based freshness analysis was designed as a supporting analysis for procedure side-effect analysis. Object freshness analysis does not feature a precise lookup and update mechanism for variables, and therefore does not offer full precision in the concrete configuration (Section 4.4.2). Moreover, the analysis (Section 4.4.2) maintains object freshness per stack-reachable application context in every state. This makes it less straightforward to evaluate the results of object freshness analysis as a standalone analysis.

However, soundness, correctness, and precision of scope-based freshness are nevertheless evaluated indirectly when testing procedure side-effect

	— ABST —		
Benchmark	Fresh/Ref	% Fresh	Time
`fib`	74 /167	44%	ε
`purity65`	74 /130	57%	ε
`purity73`	218 /518	42%	ε
`purity75`	395 /829	48%	ε
`treeadd`	8214/19876	41%	ε
`nqueens`	3102/8928	35%	ε
`dderiv`	22003/99405	22%	ε
`destruc`	10416/46942	22%	ε
`grid`	9150/24206	38%	ε
`matrix`	8948/31587	28%	ε
`fannkuch`	11946/52213	23%	ε
`mceval`	192361/4730670	4%	3”
`scm2java`	307567/3021958	10%	4”

Table 8.5: Results of running object freshness analysis under the abstract configuration. *Fresh/Ref* is the number of variables referencing fresh objects over the total number of variables, computed over all variables in all application contexts that are in the flow graph. *% Fresh* is the percentage corresponding to the previous fraction. *Time* is the running time of the abstract object freshness analysis.

analysis (Section 8.5) and purity analysis (Section 8.6). For example, if freshness analysis would incorrectly determine that a resource is fresh, so that procedure side-effect analysis would incorrectly mask a certain effect, then this error is likely to be caught in one of the many unit tests for procedure side-effect analysis.

8.4.1 Correctness

For the reasons outlined before, we performed limited correctness tests for object freshness analysis under the concrete configuration.

For a set of unit test programs, we manually specified freshness of references in all topmost application contexts arising during the concrete execution of the program. We checked that the analysis correctly determines the freshness of these references by comparing the expected outcome with the actual analysis result after joining the freshness information of all references in all topmost application contexts. We found that object freshness analysis produced the correct results for the tested programs.

8.4.2 Soundness

For soundness testing, as in our correctness tests, we only considered object freshness of references in the topmost application context. For every reference, we mechanically checked whether freshness reported by concrete object freshness analysis is subsumed by the abstract freshness result. Since both configurations use the same lattice for freshness, concrete results need not be abstracted first. Therefore, the soundness test consists in checking that the abstract freshness of a reference is not reported as fresh when its concrete freshness is reported as unfresh. We did not encounter unsound results in our set of test programs.

8.4.3 Precision

For an extensive set of small and large benchmarks, we performed precision tests of object freshness analysis in the topmost application context by counting false positives. False positives are object references that are reported as unfresh in the abstract, while being reported as fresh in the concrete.

We found no false positives, which means that abstract precision is equal to concrete precision for our benchmark programs. This is not surprising, given that object freshness analysis by design does not rely on addresses

Mnemonic	Configuration	Combination of analyses
CONC	CONC	address freshness
A	ABST	address freshness
VA	ABST	variable and address freshness
OVA	ABST	object, variable, and address freshness
EOVA	ABST	object, variable, and address freshness, combined with escape analysis

Table 8.6: Meaning of mnemonics representing different configurations and combinations of analyses.

but on lexical scopes instead, which are not affected by abstraction. Object freshness analysis is also designed as a fast but imprecise analysis, with precision loss in the concrete. In practice, the analysis' lower concrete precision is not a problem because address-based procedure side-effect analysis, which object freshness analysis aims to improve, already has full precision under the concrete configuration.

To have an indication of how many variables are determined as referencing fresh objects, Table 8.5 lists the results of running abstract object freshness analysis on our set of benchmark programs. The first observation is that the larger benchmarks tend to have low freshness percentages. Benchmark `mceval` has the lowest percentage of variables reported as referencing fresh objects. Manual inspection of the source code indicates that there are indeed many unfresh object references, but that the reported percentage of 4% is probably too low. We conjecture that the lower precision of the underlying flow analysis for this benchmark program, in combination with the limited precision of object freshness analysis, are to blame for the lower percentages.

8.4.4 Performance

Column *Time* in Table 8.5 lists the running time of abstract object freshness analysis. Relative to the running times of other individual analyses, and the overall running time (Table 8.14), object freshness analysis finishes fast. Only `mceval` (3 seconds) and `scm2java` (4 seconds) take more than a second to analyze.

8.5 Procedure Side-effect Analysis

The procedure side-effect analysis of Section 3.3 determines the observable side effects of a procedure in the context of procedure applications. The result of procedure side-effect analysis can be thought of as a database in which every program state and an application context stack-reachable in that state is linked to its set of observable effects.

Chapter 3 describes a procedure side-effect analysis using address freshness. Chapter 4 extends procedure side-effect analysis by including variable and object freshness based on scopes. Finally, Chapter 5 extends scope-based freshness so that it can be used in those application contexts on the stack for which no higher-up procedure on the stack escapes.

Table 8.6 represents the mnemonics for the different combinations of analyses we used to evaluate procedure side-effect analysis using the concrete and abstract configurations. Using the concrete configuration, we only use address freshness, so mnemonic CONC stands for both the configuration *and* concrete address-based procedure side-effect analysis.

8.5.1 Correctness

Running address-based procedure side-effect analysis under the concrete configuration results in side-effect information for application contexts that is fully precise in our approach.

We mechanically checked that extending procedure side-effect analysis with scope-based freshness and escape analysis yields identical results as in the address-based case.

We therefore conclude that, for all benchmarks, scope-based freshness analysis is correct with respect to address freshness in a concrete setting. Consequently, when evaluating soundness and precision, we only need to compare abstract results of procedure side-effect analysis with results of the CONC configuration, which only uses address freshness.

8.5.2 Soundness

For all benchmark programs, we verified that A computes observable side effects per procedure that subsume those computed using CONC after abstraction. This means that all observable side effects that occur in CONC are also reported by A. Both concrete and abstract effects are abstracted

by removing their address, and application contexts are abstracted to the applied procedure.

Next, we checked that adding scope-based freshness (VA, OVA) and escape analysis (EOVA) still yields results that subsume CONC.

From these experiments we conclude that, for all benchmarks, all combinations of abstract procedure side-effect analysis are sound: every side effect reported by the concrete analysis is also reported by every abstract analysis combination.

8.5.3 Precision

We tested the precision of abstract procedure side-effect analysis by counting the number of false positives in the resulting sets of observable side effects of all procedures that are applied in the concrete. A false positive is an effect that is reported as observable in the abstract, but for which there exists no concrete effect it subsumes, after abstraction.

Table 8.7 contains the results of this experiment. For every abstract analysis combination, we report the number of false positives (numerator) with respect to the total number of observable side effects (denominator) reported by procedure side-effect analysis, aggregated over all procedures applied in the concrete.

From the resulting percentages, we conclude that the most precise abstract combination (EOVA) is very precise for smaller benchmark programs, but less precise for larger benchmark programs. The analysis performs worst on programs `scm2java` (28%) and `dderiv` and `mceval` (39%). This is likely a consequence of the lower precision of the underlying flow analysis for these programs.

Impact of scope-based freshness

Table 8.7 shows the impact of extending procedure side-effect analysis with scope-based freshness and escape analysis on precision.

We observe that, starting from address-based observability and extending it with scope-based freshness and escape analysis, results in a reduction in the reported number of observable side effects (denominator). This means that the precision of the analysis increases, because it is able to mask more side effects.

	— A —		— VA —		— OVA —		— EOVA —	
Benchmark	fp/obs	% fp	fp/obs	% fp	fp/obs	% fp	fp/obs	% fp
`fib`	$\frac{8}{14}$	57%	$\frac{0}{6}$	0%	$\frac{0}{6}$	0%	$\frac{0}{6}$	0%
`purity65`	$\frac{4}{5}$	80%	$\frac{1}{2}$	50%	$\frac{0}{1}$	0%	$\frac{0}{1}$	0%
`purity73`	$\frac{8}{22}$	36%	$\frac{3}{17}$	18%	$\frac{3}{17}$	18%	$\frac{0}{14}$	0%
`purity75`	$\frac{18}{27}$	67%	$\frac{3}{12}$	25%	$\frac{0}{9}$	0%	$\frac{0}{9}$	0%
`treeadd`	$\frac{73}{116}$	63%	$\frac{5}{48}$	10%	$\frac{2}{45}$	4%	$\frac{2}{45}$	4%
`nqueens`	$\frac{36}{125}$	29%	$\frac{0}{89}$	0%	$\frac{0}{89}$	0%	$\frac{0}{89}$	0%
`dderiv`	$\frac{456}{881}$	52%	$\frac{271}{696}$	39%	$\frac{271}{696}$	39%	$\frac{271}{696}$	39%
`destruc`	$\frac{39}{214}$	18%	$\frac{19}{194}$	10%	$\frac{19}{194}$	10%	$\frac{19}{194}$	10%
`grid`	$\frac{75}{241}$	31%	$\frac{24}{190}$	13%	$\frac{24}{190}$	13%	$\frac{18}{184}$	10%
`matrix`	$\frac{11}{151}$	7%	$\frac{5}{145}$	3%	$\frac{5}{145}$	3%	$\frac{2}{142}$	1%
`fannkuch`	$\frac{22}{260}$	8%	$\frac{1}{239}$	0%	$\frac{1}{239}$	0%	$\frac{0}{238}$	0%
`mceval`	$\frac{12608}{31257}$	40%	$\frac{11978}{30627}$	39%	$\frac{11978}{30627}$	39%	$\frac{11961}{30610}$	39%
`scm2java`	$\frac{6131}{21124}$	29%	$\frac{5715}{20708}$	28%	$\frac{5715}{20708}$	28%	$\frac{5715}{20708}$	28%

Table 8.7: Impact of scope-based freshness analysis and escape analysis on the precision of procedure side-effect analysis. *Obs* represents the number of observable side-effects for all effects in all application contexts, as reported by procedure side-effect analysis.

However, for some benchmarks the reduction in the reported number of observable side effects is insignificant. Especially `mceval` and `scm2java` see virtually no improvements.

Also, the number of false positives (nominator) decreases by the same amount when going from A to EOVA, demonstrating that the analysis produces sound results and scope-based freshness only additionally masks false positives.

Scope-based freshness is required to have high precision in most cases. Extending procedure side-effect analysis with escape analysis has only a limited impact.

Figure 8.1 visualizes the impact on precision of extending procedure side-effect analysis with scope-based freshness and escape analysis. The data on which this chart is based can be found in Table C.1. The bar chart depicts, for each abstract analysis combination, the total number of procedure side-effects reported as observable in all states and stack-reachable application contexts (i.e., not limited to contexts for procedures that are effectively called during concrete interpretation). The lower the number of observable side-effects, the better the precision.

Figure 8.1 confirms our previous observations that scope-based freshness improves precision, but the impact of escape analysis is limited.

Impact of Abstract Garbage Collection

In our approach for determining procedure side effects, the goal of abstract garbage collection is to increase the precision of address freshness (Section 3.5), by collecting addresses that are no longer in use.

Figure 8.2 visualizes the impact of abstract garbage collection on the precision of procedure side-effect analysis. The bar chart depicts, for abstract analysis combinations A and EOVA, the total number of procedure side-effects reported as observable in all states and stack-reachable application contexts. The data on which this chart is based can be found in Table C.2.

For our least precise combination (A), we see that abstract garbage collection significantly improves precision. Address-based procedure side-effect analysis and abstract garbage collection work exclusively on store addresses, and therefore the greatest potential for improvement is for this combination.

When adding scope-based freshness and escape analysis (EOVA), procedure side-effect analysis no longer exclusively relies on address freshness, but still abstract garbage collection is beneficial in 6 out of 13 benchmarks.

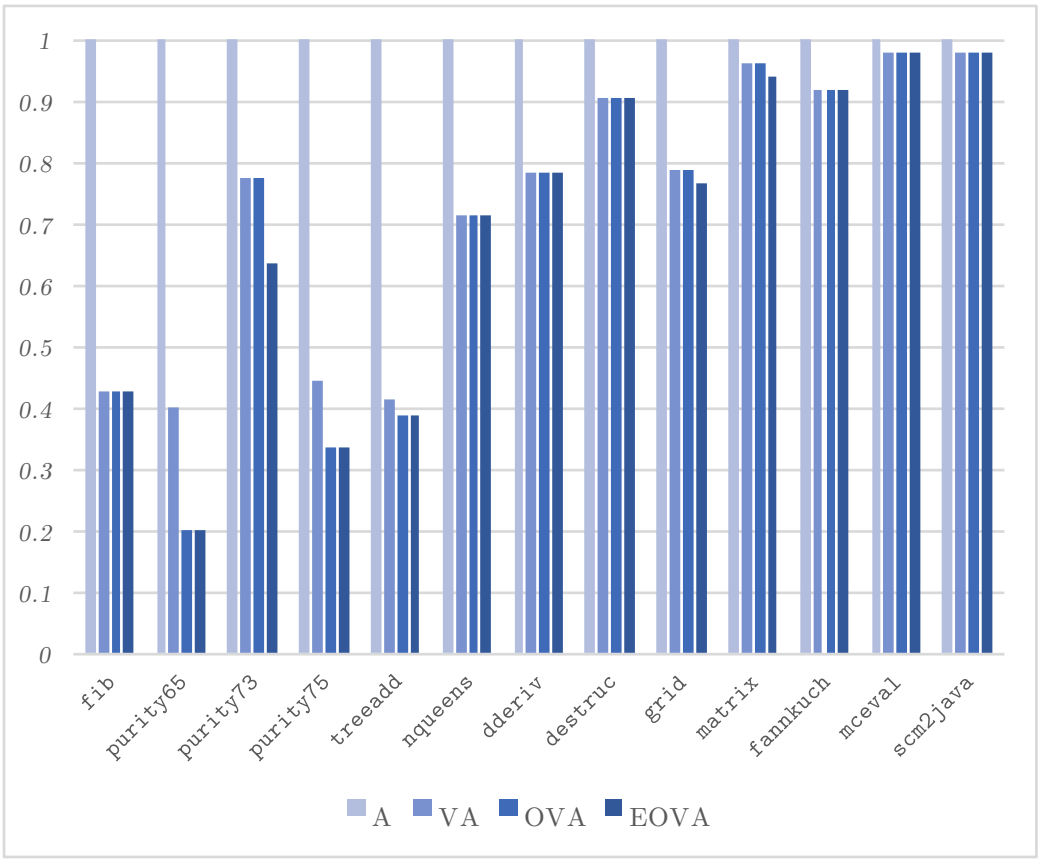


Figure 8.1: Impact of scope-based freshness analysis and escape analysis on the precision of procedure side-effect analysis. The vertical axis represents the number of reported observable procedure side effects for all application contexts in all states, normalized against A (lower is better). (Data for this chart in Table C.1 on page 213).

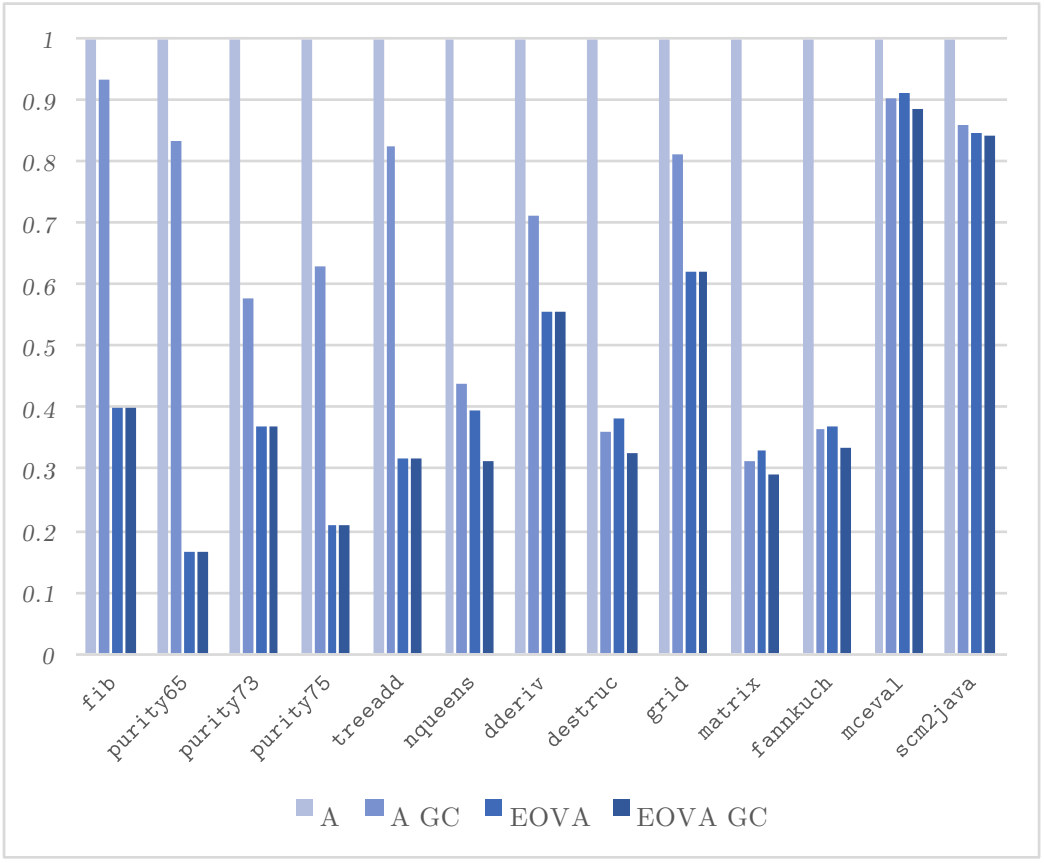


Figure 8.2: Impact of abstract garbage collection on the precision of procedure side-effect analysis using two abstract combinations A and EOVA. GC means that abstract garbage collection is enabled when determining address freshness. The vertical axis represents the number of reported observable procedure side effects for all application contexts in all states, normalized against A without GC (lower is better). (Data for this chart in Table C.2 on page 214).

Observable versus unobservable effects

So far we have evaluated precision only in terms of observable effects. We also performed an experiment to evaluate how many side effects are masked by the different combinations of procedure side-effect analysis.

Table 8.8 depicts the percentage of observable effects with respect to the total number of effects in every stack-reachable application context in every program state, if the procedure associated with the application context is applied in the concrete.

From the percentages in this table, we conclude that our procedure side-effect analysis is capable of masking side effects. For the abstract combinations, extending address-based procedure side-effect analysis with scope-based freshness and escape analysis results in a decrease of the percentage of observable side effects. This confirms that the precision of the analysis increases when going from A to EOVA, because each subsequent analysis is able to mask more side effects. However, for some benchmarks the decrease is insignificant.

Although a direct comparison between the concrete and abstract configurations is impossible, the final column shows the percentage of observable side effects under the concrete configuration. The percentage of concrete observable procedure side-effects is a rough indication of the precision of abstract procedure side-effect analysis. We observe that often precision of abstract procedure side-effect analysis is high whenever the abstract percentage is close to the concrete percentage.

8.5.4 Performance

Table 8.10 depicts the running time of procedure side-effect analysis for the different abstract combinations. We only list the running times for those benchmarks for which the overall running time (Table 8.14) is greater than one second. Comparing the duration of procedure side-effect analysis with the overall running time for the different benchmarks and combinations, we observe that procedure side-effect analysis represents between about 5% and 8% of the overall running time.

Our abstract configuration uses the P4F allocation strategy in which addresses consist of the applied procedure and the extended environment (Section 2.4.7), so caller-store analysis is required (Section 7.5). Table 8.9 lists the duration of caller-store analysis for the three benchmark programs with the longest overall running time. While caller-store analysis is not the

Benchmark	Obs A	Obs VA	Obs OVA	Obs EOVA	Obs CONC
fib	93.75%	37.5%	37.5%	37.5%	37.61%
purity65	85.71%	28.57%	14.29%	14.29%	12.5%
purity73	52.38%	40.48%	40.48%	33.33%	28.93%
purity75	60.87%	26.09%	19.57%	19.57%	22.84%
treeadd	83.55%	31.58%	29.61%	29.61%	31.2%
nqueens	41.32%	28.44%	28.44%	28.44%	28.52%
dderiv	77.08%	63.89%	63.89%	63.89%	35.53%
destruc	34.19%	30.94%	30.94%	30.94%	34.01%
grid	80.88%	65.55%	65.55%	63.66%	28.61%
matrix	29.46%	28.33%	28.33%	27.77%	35.37%
fannkuch	34.85%	32.04%	32.04%	31.9%	36.85%
mceval	93.77%	93.17%	93.17%	93.16%	35.32%
scm2java	93.49%	92.92%	92.92%	92.92%	35.69%

Table 8.8: Results of procedure side-effect analysis. *Obs* represents the percentage of observable procedure side effects, as computed by procedure side-effect analysis using the different abstract analysis combinations and the CONC configuration. The percentage is computed for all effects in all application contexts of procedures effectively applied in the concrete.

Benchmark	Time
dderiv	ε
mceval	12”
scm2java	2”

Table 8.9: Running times of caller-store analysis.

Benchmark	A	VA	OVA	EOVA
dderiv	ε	ε	ε	ε
mceval	13”	13”	13”	14”
scm2java	9”	9”	9”	14”

Table 8.10: Running times of procedure side-effect analysis for the different abstract combinations.

most expensive analysis in the set of individual analyses, timings show that its duration is not negligible for larger benchmarks. For all other programs, the running time of caller-store analysis was less than a second.

Finally, disabling abstract garbage collection during caller-store analysis significantly speeds up caller-store analysis, but the overall running time of purity analysis actually increases because of the overall reduction in precision of address freshness.

8.6 Purity Analysis

Purity analysis is the final analysis in a series of analyses, and the goal of this work (Chapter 6). Purity analysis consumes the results of side-effect analysis to determine when and where side effects occur, and the outcome of procedure side-effect analysis to determine when these side effects are observable in a particular application context. Purity analysis then summarizes the side-effecting behavior of a procedure by determining whether it generates observable side effects or depends on external side effects. This behavior can then further be classified according to different definitions of purity.

8.6.1 Correctness

For a set of unit test programs, we manually specified the expected outcome of purity analysis (sets of side-effecting behaviors **gen** and **obs**) when run under the concrete configuration. We then checked that the actual sets returned by purity analysis are identical to the expected outcome. From this we conclude that concrete address-based procedure side-effect analysis is correct for our set of unit tests.

Similar to correctness testing for procedure side-effect analysis, we checked that adding scope-based freshness and escape analysis under the concrete configuration yields identical side-effecting behavior per procedure as in the address-only case. From this we conclude that, in the concrete and for our set of benchmark programs, scope-based freshness and escape analysis are correct with respect to address-based purity analysis.

As with procedure side-effect analysis, when evaluating soundness and precision, we only need to compare the results of abstract purity analysis with results of the CONC configuration, which only uses address freshness.

8.6.2 Soundness

For all benchmark programs, we verified that the side-effecting behavior computed per procedure by A subsumes the behavior computed using CONC. This means that all side-effecting behavior reported by CONC is also reported by A. Since both the concrete and abstract configurations use the same lattice for expressing side-effecting behavior, additional abstractions are not required.

Next, we checked that adding scope-based freshness (VA, OVA) and escape analysis (EOVA) still yield abstract results that subsume CONC.

From these experiments we conclude that, for all benchmarks, all combinations of abstract purity analysis are sound: all side-effecting behavior reported by the concrete analysis is also reported by every abstract analysis combination.

8.6.3 Precision

For evaluating precision, we look at the results of purity analysis after classification. We use the the final and most precise classification scheme of Section 6.3, which distinguishes between **pure**, **observer**, and **procedure**.

Table 8.11 lists the number of procedures in each effect class as reported by both CONC and EOVA purity analysis. We choose to compare with EOVA, because this is the most precise abstract combination. Only procedures that are actually applied in the concrete are considered.

The results show that our approach is able to classify procedures according to their side-effecting behavior with high precision for small programs. Our approach has difficulty in distinguishing pure procedures from observers in three larger benchmark programs: `dderiv`, `mceval`, and `scm2java`. These are also the programs that contain the most `cons` constructs, and which exhibit the least precision in terms of control flow. Observers are characterized by the occurrence of read–write–read sequences, which are hard to detect with sufficient precision when control-flow precision is low. Under concrete semantics, the first and second read are always distinct states, while in the abstract this might not always be the case. This leads to a high number of procedures that are identified as **observer** while in fact they are **pure**.

When we take a weaker definition of purity (the first classification scheme of Section 6.3), adding **observer** to **pure**), we see that our approach is able to detect procedures that do not generate side effects with high precision.

		— CONC —			— EOVA —		
Benchmark	Called	**pure**	**obs**	**proc**	**pure**	**obs**	**proc**
`fib`	1	1	0	0	1	0	0
`purity65`	1	1	0	0	1	0	0
`purity73`	2	1	0	1	1	0	1
`purity75`	2	2	0	0	2	0	0
`treeadd`	8	8	0	0	7	0	1
`nqueens`	6	6	0	0	6	0	0
`dderiv`	10	9	0	1	3	6	1
`destruc`	11	3	1	7	3	1	7
`grid`	5	2	1	2	2	1	2
`matrix`	10	4	0	6	3	1	6
`fannkuch`	16	5	2	9	5	2	9
`mceval`	72	60	2	10	10	49	13
`scm2java`	60	58	0	2	46	3	11

Table 8.11: Absolute precision of abstract purity analysis. *Called* is the number of called procedures using the CONC configuration. **pure**, **observer**, and **procedure** counts the number of procedures in the corresponding classification. Only procedures that are effectively applied using the CONC configuration are taken into account.

Impact of scope-based freshness

Evaluating the effect of scope-based freshness and escape analysis on the precision of abstract purity analysis gives mixed results.

Table 8.12 shows that the precision of the analysis on smaller hand-crafted benchmarks (`purity65`, `purity73`, `purity75`), and existing benchmarks (`treeadd`, `grid`, `matrix`), improves by adding scope-based freshness.

On the other hand precision, mostly in the larger benchmarks (`fannkuch`, `mceval`, `scm2java`), and some smaller ones (`nqueens`, `destruc`), is not improved by scope-based freshness.

Impact of abstract garbage collection

We found that disabling abstract garbage collection results in purity analysis reporting fewer pure procedures in 4 benchmark programs (`destruc`, `matrix`, `fannkuch`, and `mceval`) when using the most precise abstract combination (EOVA).

8.6.4 Performance

Table 8.13 depicts the running time of purity analysis for the different abstract combinations. As before, we only list the running times of those benchmarks for which the overall running time (Table 8.14) exceeds one second. Classification itself is a straightforward and lightweight process, so we only report running times for the purity analysis itself.

Comparing the duration of purity analysis with the overall running time for the different benchmarks and combinations, we observe that procedure side-effect analysis represents between 68% and 78% of the overall running time. From this we conclude that the overall running time of the overall purity analysis is dominated by the final analysis, individual purity analysis itself, which maps each procedure onto a set of behaviors in terms of side effect.

8.7 Conclusion

In this chapter we evaluated our approach examining the individual analyses that comprise a purity analysis for higher-order imperative programs. We evaluated the results of the analyses in terms of correctness, soundness, precision, and performance.

		— A —			— EOVA —		
Benchmark	Called	**pure**	**obs**	**proc**	**pure**	**obs**	**proc**
`fib`	1	1	0	0	1	0	0
`purity65`	1	0	0	1	1	0	0
`purity73`	2	0	0	2	1	0	1
`purity75`	2	0	0	2	2	0	0
`treeadd`	8	3	0	5	7	0	1
`nqueens`	6	6	0	0	6	0	0
`dderiv`	10	3	6	1	3	6	1
`destruc`	11	3	1	7	3	1	7
`grid`	5	1	1	3	2	1	2
`matrix`	10	2	0	8	3	1	6
`fannkuch`	16	5	2	9	5	2	9
`mceval`	72	10	49	13	10	49	13
`scm2java`	60	46	3	11	46	3	11

Table 8.12: Impact of scope-based freshness analysis and escape analysis on the precision of purity analysis. *Called* is the number of called procedures using the CONC configuration. **pure**, **observer**, and **procedure** counts the number of procedures in the corresponding classification.

Benchmark	A	VA	OVA	EOVA
`dderiv`	1”	ε	ε	ε
`mceval`	113”	147”	147”	143”
`scm2java`	147”	115”	116”	116”

Table 8.13: Running time of individual purity analysis of the three longest-running benchmarks for the different abstract combinations.

Benchmark	A	VA	OVA	EOVA
`dderiv`	2”	1”	1”	1”
`mceval`	165”	199”	202”	199”
`scm2java`	188”	156”	161”	167”

Table 8.14: Running time of overall purity analysis of the three longest-running benchmarks for the different abstract combinations.

The experiments support the following important conclusions, stated in terms of the resulting abstract purity analysis.

- The purity analysis is capable of determining whether procedures are pure or observers in higher-order, imperative Scheme programs.
- The purity analysis produces correct results for a large set of benchmarks that express various challenges for the analysis.
- The abstract purity analysis produces results that are sound with respect to the concrete results.
- Precision of purity analysis is high for programs for which flow analysis is precisely computed. The precision of distinguishing pure procedures from observers is low for larger benchmarks in which the precision of flow analysis is low. Scope-based freshness analysis and garbage collection improves the precision of the purity analysis. Adding escape analysis to increase the opportunities for using scope-based freshness modestly improves precision in only a few benchmarks programs.
- Performance is suitable for use by clients that do not require on-the-fly results, such as on-demand static analysis tools.

In conclusion, the purity analysis realizes most of the goals we set out for it at the beginning. There are however some important limitations when it comes to the precision of detecting observers and the improvements brought by some analyses that aim to increase precision.

In the next chapter, we develop a purity analysis for JavaScript along the same lines as the one we have presented so far for Scheme. Based on the conclusions from this chapter, we made some different choices when designing the JavaScript analysis.

Chapter 9

Purity Analysis For JavaScript

9.1 Introduction

In this chapter, we transpose our techniques for determining side effects and purity to JavaScript. Like Scheme, JavaScript has both a functional and an imperative core, but also supports object-oriented programming with prototypal inheritance. The goal of porting our approach to JavaScript therefore is to evaluate if and how our ideas and techniques can be realized in a more dynamic and complex setting.

The approach we follow for JavaScript is similar to the approach for Scheme we presented in previous chapters. We focus on a core JavaScript language JS_0, and define the semantics of this language as a configurable abstract machine instrumented with effects. Based on the flow graph computed by the abstract machine for a program, we define a scope-based freshness and purity analysis.

There are, however, also important differences with our approach for Scheme. Some of these are the result of the differences in semantics between Scheme and JavaScript. Others are due to deliberate choices with regard to design and configuration of our analyses, motivated by the evaluation our purity analysis for Scheme (Section 8.7).

9.1.1 Differences with Purity Analysis for Scheme

There are a number of differences between our purity analysis for JavaScript and the one we developed for Scheme in the preceding chapters. First, there are differences stemming from the differences in the semantics between Scheme and JavaScript. Second, we also made a number of explicit design and

configuration decisions pertaining the analyses that make up our JavaScript purity analysis.

Semantic Differences

To formalize our approach, we present a core language JS_0 that represents a subset of JavaScript.

While SCHEME_0 only supported pairs, JS_0 supports objects as mappings from names to values. Conform to JavaScript semantics, each object features two dedicated fields proto and call. The former is used to implement prototypal inheritance [Crockford, 2008]. The latter enables object to be treated as function objects that can be called (the equivalent of closures).

Related to objects and prototypal inheritance, JS_0 also supports the `this` keyword and constructor calls through the `new` keyword.

Finally, JS_0 supports an explicit `return` statement.

Compared to the formalization, our proof-of-concept implementation is closer to full-fledged JavaScript with support for arrays, computed properties, the `try`–`catch` construct, and implicit type coercions (Section 9.5).

Design and Configuration Differences

Our analysis for JS_0 also differs from the analysis for SCHEME_0 in a number of design and configuration decisions.

- We use AAC as an allocation policy for stack addresses (Section 2.4.7), rather than P4F. AAC generates stack addresses that contain the caller store. Consequently, caller-store analysis (Section 7.5) is not required. Because `this` can be viewed as an implicit parameter of every method, it is also part of the application context.

- The stack store is global, but value stores are kept inside states, so there is no need for global widening (Section 7.4.1) for value stores.

- Abstract garbage collection is enabled during flow analysis. Before being added to the graph, every successor state is garbage collected. Therefore stores, including caller stores, only contain reachable addresses, and no separate reachability computation (Section 3.5.2) is required to increase the precision of address freshness.

- We implemented scope-based variable and object freshness (Chapter 4), but limited it to the topmost application context, without an extension

with escape analysis to increase the opportunities for using scope-based freshness (Chapter 5). This decision was motivated by the limited impact on precision of extending procedure side-effect analysis for SCHEME$_0$ with escape analysis (Section 8.3).

- Scope-based freshness analysis is not a separate analysis, but is implemented at the level of the effect handler, which is invoked on every effect in the flow graph during state traversal. Being limited to the topmost application context, the effect annotations produced by the abstract machine do not need to contain variables, because this information is contained in the program state.

- In addition to being restricted to the most recent application, we also simplified variable and object freshness analysis by not having it compute freshness per application context. Instead, freshness is maintained as a mapping from variables to freshness. As a consequence, no stack traversal is required when mutation occurs, because freshness of a variable represents its freshness in all application contexts.

9.1.2 Challenges

In this section we enumerate the challenges that a purity analysis for JavaScript has to overcome. Given that JavaScript shares a number of features with Scheme, such as first-class and higher-order functions, we already discussed some of the challenges in Section 6.1.3. We now discuss how these challenges manifest themselves in JavaScript, and also highlight some specific challenges that only arise for JavaScript.

The analysis still classifies functions as either **pure**, **observer**, and **procedure**, as described in Section 6.3.

Example 9.1. In the example program below, function `f` is **pure**, but `g` and `h` are not.

```
1 function f()                  // pure
2 {
3   var o={};
4   function g(p) { h(p) }      // procedure
5   function h(q) { q.x=4 }     // procedure
6   g(o)
7 }
8 f()
```

Because functions `h` (directly) and `g` (indirectly) mutate an object `o` that exists in the caller state of their applications, they are both **procedure**s. This example demonstrates that, when discovering a property write effect such as the one on line 5, the analysis has to traverse the call stack to correctly handle the effect for every function application that is active. □

While control flow is straightforward to determine in the previous example, JavaScript features higher-order functions that complicate control flow and which purity analysis must be able to handle.

Example 9.2. In the following example, function `g` is passed as an argument to function `f`.

```
1 var z=0;
2 function g(p) { z=z+1; p.x=z}     // procedure
3 function f(h) { var o={}; h(o)}   // procedure
4 f(g)
```

It is clear that function `g` is not **pure**, since it not only writes to global property `z`, but also mutates an object through its parameter. However, the purity analysis has to determine that the application of `h` in the body of `f` on line 3 applies function `g`. It therefore has to classify function `f` as a **procedure** as well, because `f` indirectly mutates `z`. □

Due to the semantics of JavaScript, it is not always straightforward to determine all effects of expressions.

Example 9.3. In the following example, variable `x` is actually a property of the global object (i.e., the value of `this` in the global scope).

```
1 var x;
2 this.x = 10; // property write effect
3 function f()
4 {
5   var y;
6   x = 10;     // property write effect
7   y = 20;     // variable write effect
8 }
```

The assignment to `x` on line 6 should be recognized as a *property* write effect on the global object, identical to the effect generated on line 2, instead of a *variable* write effect. It is important to distinguish between variables and properties because variables and objects behave and therefore influence purity differently. □

Another example of effects that are not obvious to determine, happens when an array is assigned at an index that is equal or greater than the current length of the array. This generates an additional write effect on the `length` property of that array.

JavaScript features closures, and unlike purity analyses for more traditional object-oriented languages without closures, purity analysis for JavaScript has to be able to handle free variables.

Example 9.4.

```
1  function f()                    // pure
2  {
3    function g()                  // pure
4    {
5      var z = 10;
6      function h() { z = 20 };    // procedure
7      h()
8    }
9    g()
10 }
11 f()
```

In the above example, when `z` is assigned on line 6, the call stack consists of function applications of `h`, `g`, and `f`. Function `h` is a **procedure** because it mutates `z`, which is a free variable of `h` and therefore exists in the caller state in this example. Function `g` is **pure**, because `z` is local to `g`. Function `f` is also **pure**, because `z` is not part of its scope. □

Functions that depend on external side effects are classified as observers. In order to determine which functions are observers, purity analysis needs to detect resources read by functions that are modified in between two applications.

Example 9.5. In the following example, object `o` defines a property `x` (line 1), and object `p` inherits this property through its prototype `o` (line 2).

```
1 var o = {x:123};
2 var p = Object.create(o);
3 function f()                  // observer
4 {
5   return p.x
6 }
7 f();
```

```
8 o.x = 456;
9 f()
```

Function `f` is applied twice, with a mutation of property `o.x` in between the two applications. Procedure `f` is an observer because its resulting value depends on the value of an object property that is mutated in between two applications of `f`. This example also illustrates that when property read effects are involved, we not only take the object but also the property name into account. If the property read by `f` on line 5 would be different from the property mutated on line 8, then `f` would not depend on an external side effect and be considered pure in this example. □

A final challenge we mention concerns the fact that, unlike Scheme which only features procedure application, functions in JavaScript are treated as constructors when invoked through `new`.

Example 9.6. In the following program, function `f` is invoked as a constructor (line 2) and as a regular application (line 3).

```
1 function f() { this.x = 10 }  // procedure
2 new f(); // no observable side effects
3 f();     // generates observable write effect
```

When `f` is invoked as a constructor (line 2), then it is a pure application as `this` in the body of `f` is bound to a fresh object. The regular function application (line 3) is impure because `this` is bound to the global object. As a result, function `f` is a **procedure**. □

9.1.3 Approach

Our approach for purity analysis is analogous to the the one for Scheme presented in previous chapters.

In JavaScript, observable side effects are a consequence of reading from and assigning to variables, and loading and storing object properties. Our approach for designing a purity analysis is based on an abstract interpretation of the program that integrates control flow, value flow, and effects.

We define a core imperative language JS_0 that corresponds to a representative and non-trivial subset of standard JavaScript (Section 9.2). The semantics of JS_0 is expressed as an abstract machine that transitions between states. The abstract machine generates appropriate read and write effects caused by reading from and writing to variables and object properties.

Purity analysis then examines the flow graph to determine whether a given function can be classified as pure, observer, or procedure by considering all effects during and in between applications of the function (Section 9.3). For every effect, the analysis traverses the call stack to obtain all active function applications at that point. When in the context of an application an observable write effect is generated, then the applied function becomes a procedure. For dependence on external effects, purity analysis looks for read–write–read sequences in the program involving the same resource, with the initial and subsequent read occurring in an application of the same function.

The purity analysis uses scope-based freshness of variables and object references to mask certain read and write effects, thereby increasing the precision of the purity analysis. Compared to the object freshness analysis for Scheme (Section 4.4.2), we simplified the formulation of object freshness analysis incorporated in our purity analysis for JavaScript. Using variable and object freshness is explicitly limited it to the application context on top of the stack in the program state at which the effect occurs. This simplification improves running times, yet does not seem to result in a loss of precision for our benchmark programs.

After explaining our approach in detail, the remainder of this chapter discusses our implementation (Section 9.5) and experiments (Section 9.6).

9.1.4 Contributions

The contributions presented in this chapter are the following:

- We present an abstract machine for a core JavaScript-like language JS_0 that tracks read and write effects generated by accessing and modifying variables and object properties during interpretation (Section 9.2.2).

- We introduce a purity analysis for JS_0 over a flow graph annotated with effects (Section 9.3).

- We define a scope-based freshness analysis for variables and objects to improve the precision of this purity analysis (Section 9.4).

- We implement a purity analysis for a substantial subset of JavaScript (Section 9.5), and experiment with it on several JavaScript benchmarks (Section 9.6).

9.2 Setting

Because function call and return is the dominant pattern in higher-order, functional programs, an analysis needs to model call/return precisely. For this reason, we use a pushdown analysis [Johnson and Van Horn, 2014] and not a more classic finite-state analysis.

9.2.1 Input Language

In order to simplify the formalization of our approach, we work on JS_0, a core functional language with assignment. JS_0, depicted in Figure 9.1, most notably features objects as maps, higher-order functions, prototypal inheritance, and assignment.

Like our Scheme input language SCHEME_0, JS_0 distinguishes between simple (atomic) and compound expressions, and only allows simple expressions in most positions where regular JavaScript allows any kind of expression. Functions are not simple expressions, because they evaluate to an address pointing to a function *objects* in the store, and thus require store allocation.

Although JS_0 is a small language, its set of features is sufficiently challenging for performing purity analysis. Our implementation (Section 9.5), used to validate our approach, supports a larger subset of traditional features like variable declarations, iteration, non-local return flow, and typical features of JavaScript, including type coercions and parts of the standard built-in functions and objects.

9.2.2 Semantics

The small-step semantics of JS_0 is expressed as an abstract machine [Felleisen and Friedman, 1987] that transitions between evaluation (**ev**) and continuation (**ko**) states. The resulting machine is another variation on the CESIK$^\star\Xi$ abstract machine described in Johnson and Van Horn [Johnson and Van Horn, 2014]. This machine also is an *abstract* abstract machine since it operates on abstract values (Section 2.4.2), although it can be parameterized to express concrete semantics (Section 2.4.3).

In what follows, we further detail the operation of the abstract machine.

State Space

Figure 9.2 shows the abstract state-space. The control (e), environment (ρ), store (σ), and value (d) components of the machine are standard (see

$$
\begin{array}{rll}
e \in \mathsf{Exp} ::= & s & \text{[simple expr]} \\
| & f & \text{[function]} \\
| & v\texttt{(}s\texttt{)} & \text{[function call]} \\
| & s_0\texttt{.}v\texttt{(}s_1\texttt{)} & \text{[method call]} \\
| & \texttt{new } v\texttt{(}s\texttt{)} & \text{[new expr]} \\
| & v\texttt{=}e & \text{[assignment]} \\
| & s\texttt{.}v & \text{[property load]} \\
| & s\texttt{.}v\texttt{=}e & \text{[property store]} \\
| & \texttt{return } s & \text{[return]} \\
s \in \mathsf{Simple} ::= & v & \text{[variable]} \\
| & \texttt{this} & \text{[this expr]} \\
f \in \mathsf{Fun} ::= & \lambda\texttt{(}v\texttt{)\{}e\texttt{\}} & \\
v \in \mathsf{Var} = & \text{a set of identifiers} &
\end{array}
$$

Figure 9.1: Input language JS_0.

Section 2.4). As in the abstract machine for SCHEME_0, stacks consist of a local continuation (ι) delimited by a meta-continuation (κ). The local continuation is a (possibly empty) list of frames, while the meta-continuation is a calling context. Calling contexts are generated at call sites, except for the root calling context that is created at the start of program evaluation.

Calling contexts serve as stack addresses pointing to underlying stacks that are stored in a stack store (Ξ). For the semantics of JS_0 we use the AAC stack address allocator (Section 2.4.7), generating stack address that contain five components: a call expression (e), a callable (c), an argument (d_{arg}), a `this` pointer (a_{this}), and a caller store that is in effect at function entry (σ). Allocating stacks with this kind of precision describes unbounded stacks in a finite way with maximally attainable call/return precision depending on the employed abstractions, and full precision under concrete semantics.

Value $d_{\text{undefined}}$ represents the unique value for `undefined`.

Objects are represented as maps from properties to values. Properties call and prototype are two special properties that are distinct from all other properties, and are used to implement function objects and object prototypes, respectively. Compared to JavaScript semantics, the handling of

object property lookup is simplified in JS_0. JS_0 does not distinguish between properties that must be present and properties that may be present. The latter case arises after joining two objects, in which only one of the objects has a particular property.

Program Injection

The injection function $\mathcal{I} : \mathsf{Exp} \to \mathit{State}$ turns an expression into an initial evaluation state with empty environment, initial store σ_0, empty local continuation, and the root context κ_0 as meta-continuation.

$$\begin{aligned} \mathcal{I}(e) &= \mathbf{ev}(e, [], \sigma_0, \langle\rangle, \kappa_0, []) \\ \text{where } \kappa_0 &= (e, \bot, \bot, a_0, \sigma_0) \\ \sigma_0 &= [a_0 \mapsto []] \end{aligned}$$

The initial store σ_0 maps the global object at address a_0, which we assume to be globally available.

Address Allocation

Address allocation is a parameter of the semantics that can be used to control the context-sensitivity of the resulting analysis. We assume the presence of allocation functions *allocVar* for allocating variables, *allocCtr* for allocating constructor objects, *allocFun* for allocating function objects, and *allocProto* for allocating prototypes of objects. Any address allocation scheme is sound [Might and Manolios, 2009], although to simplify the semantics we assume that addresses for objects (in *Obj*) are distinct from addresses for all other values (in D).

To express concrete semantics, we can take $\mathit{Addr} = \mathbb{N}$ and

$$\mathit{allocX}(e, \rho, \sigma, \iota, \kappa) = 1 + \max(\text{Dom}(\sigma)),$$

where *allocX* is one of the allocation functions and Dom returns the domain of a function.

For abstract semantics, a monovariant allocation scheme (0CFA) would be $\mathit{Addr} = \mathsf{Exp}$ with

$$\mathit{allocVar}(e, \rho, \sigma, \iota, \kappa) = e,$$

and similar definitions for the other allocators.

$$
\begin{array}{rll}
\varsigma \in \mathit{State} & ::= \mathbf{ev}(e, \rho, \sigma, \iota, \kappa, \Xi) & \text{[eval state]} \\
& \mid \ \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi) & \text{[kont state]} \\
\rho \in \mathit{Env} & = \mathsf{Var} \rightharpoonup \mathit{Addr} & \text{[environment]} \\
\sigma \in \mathit{Store} & = \mathit{Addr} \rightharpoonup (D + \mathit{Obj}) & \text{[store]} \\
d \in D & = \mathcal{P}(\mathit{Addr} + d_{\text{undefined}}) & \text{[value]} \\
\omega \in \mathit{Obj} & = (\mathsf{Var} \rightharpoonup D) & \text{[object]} \\
& \quad \times (\text{proto} \mapsto D) & \\
& \quad \times (\text{call} \mapsto \mathcal{P}(\mathit{Callable})) & \\
c \in \mathit{Callable} & ::= (f, \rho) & \text{[callable]} \\
\iota \in \mathit{LKont} & = \mathit{Frame}^* & \text{[frame]} \\
\phi \in \mathit{Frame} & ::= \mathbf{as}(v, \rho) & \text{[assignment frame]} \\
& \mid \ \mathbf{st}(s, v, \rho) & \text{[property store frame]} \\
\kappa \in \mathit{Kont} & ::= (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma) & \text{[meta-continuation]} \\
\Xi \in \mathit{KStore} & = \mathit{Kont} \rightharpoonup \mathcal{P}(\mathit{LKont} \times \mathit{Kont}) & \text{[stack store]} \\
a \in \mathit{Addr} & \text{ is a set of addresses} & \text{[address]} \\
\mathit{eff} \in \mathit{Eff} & ::= \mathbf{Wv}(a, v) & \text{[variable write effect]} \\
& \mid \ \mathbf{Wf}(a, v) & \text{[property write effect]} \\
& \mid \ \mathbf{Rv}(a, v) & \text{[variable read effect]} \\
& \mid \ \mathbf{Rf}(a, v) & \text{[property read effect]} \\
E \in \mathcal{P}(\mathit{Eff}) & \text{ is a set of effects} & \text{[effects]}
\end{array}
$$

Figure 9.2: Abstract state-space of the flow analysis.

Simple Expressions

Simple expressions are the equivalent of atomic expressions in SCHEME$_0$ semantics. The primary difference with SCHEME$_0$ is that functions do not atomically evaluate to a closure, but to an equivalent function object in which the call property contains the closure. Because function evaluation allocates objects (the function object and its prototype), it cannot be considered as a simple expression.

Simple expressions in our JavaScript semantics are variables and `this` expressions. Function $\mathit{evalSimple} : \mathsf{Simple} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \mapsto D \times \mathcal{P}(\mathit{Eff})$ evaluates simple expressions and returns the resulting value, as well as the set of generated effects.

To evaluate a variable reference, first the address of the variable is looked up in the lexical environment. Then the value associated with that address in the store is returned, and a variable read effect is generated.

$$\begin{gathered}\mathit{evalSimple}(v, \rho, \sigma, \kappa) = (\sigma(a), \{\mathbf{Rv}(a, v)\}) \\ \text{if } v \in \mathrm{Dom}(\rho) \\ \text{where } a = \rho(v)\end{gathered}$$

If the variable is not available in the environment, a property lookup is performed on the global object at address a_0, and a property read effect is generated.

$$\begin{gathered}\mathit{evalSimple}(v, \rho, \sigma, \kappa) = (\omega(v), \{\mathbf{Rf}(a_0, v)\}) \\ \text{where } \omega = \sigma(a_0)\end{gathered}$$

The value for a `this` expression is retrieved from the current calling context, and no effects are generated.

$$\mathit{evalSimple}([\![\texttt{this}]\!], \rho, \sigma, (e, c, d_{\mathrm{arg}}, a_{\mathrm{this}}, \sigma)) = (a_{\mathrm{this}}, \varnothing)$$

Property Lookup

Relation *lookupProp* looks up a property by traversing the prototype chain of an object. If the property is not found in the chain, `undefined` is returned.

$$\begin{aligned} &\mathit{lookupProp}(v, a, \sigma) \\ &\qquad = \begin{cases} (\omega(v), \{\mathbf{Rf}(a, v)\}) & \text{if } v \in \mathrm{Dom}(\omega) \\ (\{d_{\text{undefined}}\}, \varnothing) & \text{if } \omega(\text{proto}) = \varnothing \\ \mathit{lookupProp}(v, a', \sigma) & \text{else} \end{cases} \\ &\text{where } \omega = \sigma(a) \\ &\qquad a' \in \omega(\text{proto}) \end{aligned}$$

Function Call

Function *evalCall* applies a function to an argument in a given context. It extends the procedure's static environment by binding the argument, and moves evaluation to the body of the function. Parameter κ is the application context of the caller, while parameter κ' represents the application context for the call itself.

$$\begin{aligned} \mathit{evalCall}((f, \rho), d_{\text{arg}}, \sigma, \iota, \kappa, \Xi, \kappa') &= \mathbf{ev}(e, \rho', \sigma', \langle\rangle, \kappa', \Xi') \\ \text{where } f &= [\![\lambda(v)\{e\}]\!] \\ \rho' &= \rho[v \mapsto a] \\ \sigma' &= \sigma \sqcup [a \mapsto d_{\text{arg}}] \\ a &= \mathit{allocVar}(v, \rho, \sigma, \iota, \kappa) \\ \Xi' &= \Xi \sqcup [\kappa' \mapsto \{(\iota, \kappa)\}] \end{aligned}$$

Transition Relation

In order to determine function purity, we need to be able to reason about effects that occur as a result of reading and mutating variables and object properties during evaluation. We make write effects explicit by modeling them on the transition relation that transitions between states: $(\longmapsto) \sqsubseteq \mathit{State} \times \mathit{State} \times \mathcal{P}(\mathit{Eff})$.

1. A simple expression is evaluated by delegating to *evalSimple*.

$$\begin{aligned} \mathbf{ev}(s, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto (\mathbf{ko}(d, \sigma, \iota, \kappa, \Xi), E) \\ \text{where } (d, E) &= \mathit{evalSimple}(s, \rho, \sigma, \kappa) \end{aligned}$$

2. Evaluating a function expression yields a reference to a function object (ω_f) that is allocated in the store. Following JavaScript semantics, a function object has a fresh object assigned to its `prototype` property.

$$\begin{aligned}
\mathbf{ev}(\overbrace{[\![\texttt{λ(}v\texttt{)\{}e\texttt{\}}]\!]}^{f}, \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto (\mathbf{ko}(\{a\}, \sigma', \iota, \kappa, \Xi), \varnothing) \\
\text{where } a &= \mathit{allocFun}(f, \rho, \sigma, \iota, \kappa) \\
a' &= \mathit{allocProto}(f, \rho, \sigma, \iota, \kappa) \\
\sigma' &= \sigma \sqcup [a \mapsto \omega_f, a' \mapsto \omega_{\text{proto}}] \\
\omega_f &= [\text{call} \mapsto \{(f, \rho)\}, \\
&\qquad \text{proto} \mapsto \varnothing \\
&\qquad \texttt{prototype} \mapsto \{a'\}] \\
\omega_{\text{proto}} &= [\text{proto} \mapsto \varnothing]
\end{aligned}$$

3. A function call is evaluated by first evaluating operator and argument, and then applying the *evalCall* helper function with a reference to the global object (a_0) as `this` value.

$$\begin{aligned}
\mathbf{ev}(\overbrace{[\![v\texttt{(}s\texttt{)}]\!]}^{e}, \rho, \sigma_\tau, \iota, \kappa, \Xi) &\longmapsto (\mathit{evalCall}(c, d_{\text{arg}}, \sigma_\tau, \iota, \kappa, \Xi, \kappa'), E) \\
\text{where } (d_f, E_f) &= \mathit{evalSimple}(v, \rho, \sigma_\tau, \kappa) \\
(d_{\text{arg}}, E_{\text{arg}}) &= \mathit{evalSimple}(s, \rho, \sigma_\tau, \kappa) \\
a_f &\in d_f \\
\omega_f &= \sigma_\tau(a_f) \\
c &\in \omega_f(\text{call}) \\
\kappa' &= (e, c, d_{\text{arg}}, a_0, \sigma_\tau) \\
E &= E_f \cup E_{\text{arg}}
\end{aligned}$$

4. For a method call additionally the method is looked up in the receiver, and the receiver is set as value for `this` in the new calling context.

$$\begin{aligned}
\mathbf{ev}(\overbrace{[\![s_0\,.\,v(s_1)]\!]}^{e}, \rho, \sigma_\tau, \iota, \kappa, \Xi) &\longmapsto (\mathit{evalCall}(c, d_{\text{arg}}, \sigma_\tau, \iota, \kappa, \Xi, \kappa'), E) \\
\text{where } (d_{\text{this}}, E_{\text{this}}) &= \mathit{evalSimple}(s_0, \rho, \sigma_\tau, \kappa) \\
(d_{\text{arg}}, E_{\text{arg}}) &= \mathit{evalSimple}(s_1, \rho, \sigma_\tau, \kappa) \\
a_{\text{this}} &\in d_{\text{this}} \\
(d_f, E_f) &\in \mathit{lookupProp}(v, a_{\text{this}}, \sigma_\tau) \\
a_f &\in d_f \\
\omega_f &= \sigma_\tau(a_f) \\
c &\in \omega_f(\text{call}) \\
\kappa' &= (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma_\tau) \\
E &= E_{\text{this}} \cup E_{\text{arg}} \cup E_f
\end{aligned}$$

5. A constructor call allocates a new object on the heap, and sets a reference to this object as value for `this` in the new calling context. The internal prototype of the new object is the value of the `prototype` property of the invoked constructor. The caller store in the context is the store *without* the newly created object.

$$\begin{aligned}
\mathbf{ev}(\overbrace{[\![\mathsf{new}\ v(s)]\!]}^{e}, \rho, \sigma_\tau, \iota, \kappa, \Xi) &\longmapsto (\mathit{evalCall}(c, d_{\text{arg}}, \sigma', \iota, \kappa, \Xi, \kappa'), E) \\
\text{where } (d_f, E_f) &= \mathit{evalSimple}(v, \rho, \sigma_\tau, \kappa) \\
(d_{\text{arg}}, E_{\text{arg}}) &= \mathit{evalSimple}(s, \rho, \sigma_\tau, \kappa) \\
a_f &\in d_f \\
\omega_f &= \sigma_\tau(a_f) \\
c &\in \omega_f(\text{call}) \\
a_{\text{this}} &= \mathit{allocCtr}(e, \rho, \sigma_\tau, \iota, \kappa) \\
\omega &= [\text{proto} \mapsto \omega_f(\mathtt{prototype})] \\
\sigma' &= \sigma_\tau \sqcup [a_{\text{this}} \mapsto \omega] \\
\kappa' &= (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma_\tau) \\
E &= E_f \cup E_{\text{arg}}
\end{aligned}$$

6. Variable assignment pushes a continuation to assign the value of the right hand side to the variable. No effects are generated during this

step.

$$\begin{aligned}\mathbf{ev}([\![v{=}e]\!], \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto (\mathbf{ev}(e, \rho, \sigma, \phi : \iota, \kappa, \Xi), \varnothing) \\ &\text{where } \phi = \mathbf{as}(v, \rho)\end{aligned}$$

7. Loading a property involves evaluating the receiver, and looking up the property in that receiver.

$$\begin{aligned}\mathbf{ev}([\![s\,.\,v]\!], \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto (\mathbf{ko}(d, \sigma, \iota, \kappa, \Xi), E) \\ &\text{where } (d_r, E_r) = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \\ &\qquad a \in d_r \\ &\qquad (d, E') \in \mathit{lookupProp}(v, a, \sigma) \\ &\qquad E = E_r \cup E'\end{aligned}$$

8. Like assignment, storing a property requires evaluating the right hand side and pushing a continuation to perform the actual property update.

$$\begin{aligned}\mathbf{ev}([\![s\,.\,v{=}e]\!], \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto (\mathbf{ev}(e, \rho, \sigma, \phi : \iota, \kappa, \Xi), \varnothing) \\ &\text{where } \phi = \mathbf{st}(s, v, \rho)\end{aligned}$$

9. Function return computes a return value and clears the local continuation.

$$\begin{aligned}\mathbf{ev}([\![\texttt{return}\ s]\!], \rho, \sigma, \iota, \kappa, \Xi) &\longmapsto (\mathbf{ko}(d, \rho, \sigma, \langle\rangle, \kappa, \Xi), E) \\ &\text{where } (d, E) = \mathit{evalSimple}(s, \rho, \sigma, \kappa)\end{aligned}$$

10. When the machine has to continue with an assignment frame on top of the stack, it assigns the value computed for the right hand side to the variable on the left, if the variable is in scope. It then continues with this value, generating a variable write effect.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \mathbf{as}(v, \rho) : \iota, \kappa, \Xi) &\longmapsto (\mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), E) \\ &\qquad \text{if } v \in \mathrm{Dom}(\rho) \\ &\text{where } a = \rho(v) \\ &\qquad \sigma' = \sigma \sqcup [a \mapsto d] \\ &\qquad E = \{\mathbf{Wv}(a, v)\}\end{aligned}$$

11. If the variable is not found in the environment, the machine performs a property update on the global object, generating a property write effect.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \mathbf{as}(v, \rho) : \iota, \kappa, \Xi) \longmapsto (\mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), E) \\ \text{where } \omega = \sigma(a_0)[v \mapsto d] \\ \sigma' = \sigma \sqcup [a_0 \mapsto \omega] \\ E = \{\mathbf{Wf}(a_0, v)\}\end{aligned}$$

12. Storing a property always happens directly on the receiver and does not require traversing prototype links. It generates a property write effect.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \mathbf{st}(s, v, \rho) : \iota, \kappa, \Xi) \longmapsto (\mathbf{ko}(d, \sigma', \iota, \kappa, \Xi), E) \\ \text{where } (d_r, E_r) = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \\ a \in d_r \\ \omega = \sigma(a)[v \mapsto d] \\ \sigma' = \sigma \sqcup [a \mapsto \omega] \\ E = E_r \cup \{\mathbf{Wf}(a, v)\}\end{aligned}$$

13. When the machine reaches a state with an empty local continuation, the machine dereferences the stack address to obtain an underlying stack. If no stacks are found in the stack store, then the machine has reached a program exit and halts, and the current value is the result value of the program. Else, the machine has reached a function exit. When exiting a constructor call, a reference to the newly created object is returned.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) \longmapsto (\mathbf{ko}(d', \sigma, \iota', \kappa', \Xi), \varnothing) \\ \text{where } (\iota', \kappa') \in \Xi(\kappa) \\ d' = \{a_{\text{this}}\} \\ ([\![\texttt{new } v\texttt{(}s\texttt{)}]\!], \_, \_, a_{\text{this}}, \_) = \kappa\end{aligned}$$

14. When exiting a regular function call, the current value is returned.

$$\begin{aligned}\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) \longmapsto (\mathbf{ko}(d, \sigma, \iota', \kappa', \Xi), \varnothing) \\ \text{where } (\iota', \kappa') \in \Xi(\kappa)\end{aligned}$$

Flow Graph

We determine function purity by reasoning about effects that happen during program evaluation. We therefore construct a *flow graph* representing program evaluation, in which nodes are reachable states, and edges are transitions between states that are labeled with the effects that occur on transition. Let $\hookrightarrow$ be transition relation $\longmapsto$ with the effects removed: $\varsigma \hookrightarrow \varsigma' \iff \varsigma \longmapsto (\varsigma', E)$. Evaluation can be expressed as computing the transitive closure of $\hookrightarrow$ after injection.

$$\mathcal{E}(e) = \{\varsigma \mid \mathcal{I}(e) \hookrightarrow^* \varsigma'\}$$

The definition of flow graph G_e for expression e then is as follows:

$$\varsigma \xrightarrow{E} \varsigma' \in G_e \iff \varsigma \in \mathcal{E}(e) \text{ and } \varsigma \longmapsto (\varsigma', E)$$

Static analysis requires a finite flow graph for every possible program. We can guarantee finiteness by plugging in finite sets for Var and *Addr* into the state-space of the analysis (Figure 9.2). For finite programs the entire state space is then finite as well, and $\hookrightarrow$, which is monotonic, has a least fixpoint.

9.3 Purity Analysis

Using the flow graph from the previous section, we are able to determine the purity of a function by examining all of its applications. This examination has to happen along two dimensions: we have to traverse all states (Section 9.3.1), and in every state we have to traverse the call stack (Section 9.3.2) because every effect occurs for every active function. The result is a map P from functions to their effect class.

$$\begin{aligned}
\mathit{class} \in \mathit{Class} &= \{\mathbf{pure}, \mathbf{observer}, \mathbf{procedure}\} && \text{[effect class]} \\
P \in \mathit{Purity} &= \mathsf{Fun} \mapsto \mathit{Class} && \text{[function purity]} \\
R \in \mathit{Read} &= \mathit{Res} \mapsto \mathcal{P}(\mathsf{Fun}) && \text{[read table]} \\
O \in \mathit{Obs} &= \mathit{Res} \mapsto \mathcal{P}(\mathsf{Fun}) && \text{[observer table]} \\
\mathit{res} \in \mathit{Res} &= \mathit{Addr} + (\mathit{Addr} \times \mathsf{Var}) && \text{[resource]}
\end{aligned}$$

Following the classification scheme of Gifford and Lucassen [1986] (Section 6.3), the effect class (*Class*) is a join semi-lattice in which $\bot \sqsubset \mathbf{pure} \sqsubset \mathbf{observer} \sqsubset \mathbf{procedure}$, so that for example $\mathbf{pure} \sqcup \mathbf{observer} = \mathbf{observer}$ and $\mathbf{pure} \sqcup \mathbf{procedure} = \mathbf{procedure}$. The effect class of a function is

procedure if an application of that function generates observable side effects, **observer** if an application depends on external side effects but none generate observable side effects, and **pure** otherwise.

Purity analysis navigates the flow graph G_e, and updates maps P, R and O. Map P maps functions onto their effect class, and initially every applied function is **pure**. The effect class of unapplied functions remains $\bot$.

When during an application an observable write effect is generated, then the applied function becomes a procedure (**procedure**) in P.

For dependence on external effects, purity analysis looks for read–write–read sequences involving the same resource, with the initial and subsequent read occurring in an application of the same function. If a function reads an external resource, this is tracked in the read table (R). If that resource is mutated, then all functions that have a read dependency on that resource are added to the observer table (O). If a function reads a resource, and is mapped in the observer table as a potential observer for that resource, then that function becomes an actual observer of an external effect, and its effect class at that point is joined with **observer** in P.

As before, the analysis considers variables and object properties to be resources (Res) on which read and write effects are possible.

The purity analysis we present in this section is finite if the underlying flow graph is finite, because the purity maps monotonically increase in a finite domain.

9.3.1 Graph Traversal

Traversal over a flow graph G_e is handled by $travGraph_P$, which delegates every newly encountered state to *handleEffect* and joins the result back in the maps P, R, and O. If any of the maps are updated by *handleEffect*, then the set of visited states is cleared to ensure we visit every state with

the maximal configuration of these maps.

$$\begin{aligned}
\mathit{travGraph}_P(S, \varnothing, P, R, O) &= P \\
\mathit{travGraph}_P(S \cup \{\varsigma\}, W \uplus \{\varsigma\}, P, R, O) &= \mathit{travGraph}_P(S \cup \{\varsigma\}, W, P, R, O) \\
\mathit{travGraph}_P(S, W \uplus \{\varsigma\}, P, R, O) &= \mathit{travGraph}_P(S', W', P', R', O') \\
\text{where } (P', R', O') &= \bigsqcup \{\mathit{handleEffect}(\varsigma, \mathit{eff}, P, R, O) \\
&\qquad \mid \mathit{eff} \in E \wedge (\varsigma \xrightarrow{E} \varsigma') \in G_e\} \\
W' &= W \cup \{\varsigma' \mid (\varsigma \to \varsigma') \in G_e\} \\
S' &= \begin{cases} \varnothing & \text{if } P \neq P' \vee R \neq R' \vee O \neq O' \\ S \cup \{\varsigma\} & \text{else} \end{cases}
\end{aligned}$$

The state handler dispatches to *handleWrite* and *handleRead* after building the resource.

$$\mathit{handleEffect}(\varsigma, \overbrace{\mathbf{Wv}(a, \_)}^{\mathit{eff}}, P, R, O) = \mathit{handleWrite}(\mathit{eff}, a, \varsigma, P, R, O)$$

$$\mathit{handleEffect}(\varsigma, \overbrace{\mathbf{Wf}(a, v)}^{\mathit{eff}}, P, R, O) = \mathit{handleWrite}(\mathit{eff}, (a, v), \varsigma, P, R, O)$$

$$\mathit{handleEffect}(\varsigma, \overbrace{\mathbf{Rv}(a, \_)}^{\mathit{eff}}, P, R, O) = \mathit{handleRead}(\mathit{eff}, a, \varsigma, P, R, O)$$

$$\mathit{handleEffect}(\varsigma, \overbrace{\mathbf{Rf}(a, v)}^{\mathit{eff}}, P, R, O) = \mathit{handleRead}(\mathit{eff}, (a, v), \varsigma, P, R, O)$$

When a read effect is encountered, the stack is traversed to update the maps R (in case of a new read effect) and P (in case of a read-write-read sequence being detected).

$$\begin{aligned}
\mathit{handleRead}(\mathit{eff}, \mathit{res}, \varsigma, P, R, O) &= (P', R', O) \\
\text{where } (P', R') &= \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, \varnothing, \{\kappa\}, P, R, O) \\
(\ldots, \kappa, _) &= \varsigma
\end{aligned}$$

When a write effect is encountered, the stack is traversed to update the map P, and functions performing read operations on the resources are joined in

the O map.

$$\begin{aligned} \mathit{handleWrite}(\mathit{eff}, \mathit{res}, \varsigma, P, R, O) &= (P', R, O') \\ \text{where } O' &= O \sqcup [\mathit{res} \mapsto R(\mathit{res})] \\ P' &= \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, \varnothing, \{\kappa\}, P) \\ (\ldots, \kappa, \_) &= \varsigma \end{aligned}$$

9.3.2 Stack Traversal

Purity analysis needs to determine observability of an effect for every function that is computing a return value at the point at which the effect is produced (Section 3.3.2). For this it needs to find all the active application contexts in which the effect occurs, and does so by traversing the stack. Because continuations are delimited by applications, the analysis only has to look at meta-continuations.

The stack-traversing operation $\mathit{travStack}_R$ propagates a read effect down the stack, while remembering the application contexts it has already seen to avoid infinite recursion. The caller store is used to check whether effects that occur during function application are observable or not.

1. Stack traversal terminates when the work list is empty. The purity and read maps are returned.

$$\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S, \varnothing, P, R, O) = (P, R)$$

2. An application context that was already visited, is not visited again.

$$\begin{aligned} &\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W \uplus \{\kappa\}, P, R, O) \\ &\qquad = \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W, P, R, O) \end{aligned}$$

3. If a read effect occurs on an address that is mapped in the caller store, then the effect is on a resource that is reachable by the caller. In this case the function is marked as read-dependent on the resource (by adding it to R). If the function being applied is registered as a potential observer for the read resource, then the function is marked

as an observer in P.

$$\begin{aligned}
&\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\kappa\}, P, R, O) \\
&\qquad = \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S', W', P', R', O) \\
&\qquad \text{if } a \in \mathrm{Dom}(\sigma_\tau) \\
&\qquad \text{where } S' = S \cup \{\kappa\} \\
&\qquad\qquad W' = W \cup \{\kappa' \mid (_, \kappa') \in \Xi(\kappa)\} \\
&\qquad\qquad P' = P \sqcup \bigsqcup \{f \mapsto \mathbf{observer} \mid f \in O(\mathit{res})\} \\
&\qquad\qquad R' = R \sqcup [\mathit{res} \mapsto \{f\}] \\
&\qquad\qquad (\ldots, \Xi) = \varsigma \\
&(_, (f, _), _, _, \sigma_\tau) = \kappa
\end{aligned}$$

4. If the read address is not in the domain of the caller store, then the effect is local to the application and can be masked.

$$\begin{aligned}
&\mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\kappa\}, P, R, O) \\
&\qquad = \mathit{travStack}_R(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W, P, R, O)
\end{aligned}$$

Function $\mathit{travStack}_W$ similarly traverses the stack, using the caller store to check whether write effects are observable or not.

1. Stack traversal terminates when the work list is empty. The purity map is returned.

$$\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S, \varnothing, P) = P$$

2. An application context that was already visited, is not visited again.

$$\begin{aligned}
&\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W \uplus \{\kappa\}, P) \\
&\qquad = \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W, P)
\end{aligned}$$

3. If the address of the written resource is mapped in the caller store of an active function application, then the effect is observable from the point of view of the caller, and the function is marked as a procedure

in P.

$$\begin{aligned}
&\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\kappa\}, P) \\
&\qquad = \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S', W', P') \\
&\qquad \text{if } a \in \mathrm{Dom}(\sigma_\tau) \\
&\qquad \text{where } S' = S \cup \{\kappa\} \\
&\qquad\quad W' = W \cup \{\kappa' \mid (_, \kappa') \in \Xi(\kappa)\} \\
&\qquad\quad P' = P \sqcup [f \mapsto \textbf{procedure}] \\
&\qquad\quad (\ldots, \Xi) = \varsigma \\
&\qquad (_, (f, _), _, _, \sigma_\tau) = \kappa
\end{aligned}$$

4. If the written address is not in the domain of the caller store, then the effect is local to the application and can be masked.

$$\begin{aligned}
&\mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S, W \uplus \{\kappa\}, P) \\
&\qquad = \mathit{travStack}_W(\mathit{eff}, \mathit{res}, \varsigma, S \cup \{\kappa\}, W, P)
\end{aligned}$$

9.4 Freshness Analysis

The address-based purity analysis for JS_0 presented in the previous section, suffers from the same drawback as the one we identified for address-based observability for SCHEME_0 (Section 4.2.1). When the abstract machine from Section 9.3 is configured with a concrete allocator, addresses are generated with full precision. As a result, purity analysis will determine function purity with full precision as well, without false positives or negatives. However, to guarantee that the analysis runs in finite time and space, we sacrifice full precision primarily by allowing the abstract machine to allocate addresses that are already in use while it is evaluating the input program. As was the case for SCHEME_0, this address allocation scheme results in a loss of precision for address-based purity analysis, as Example 9.7 illustrates.

Example 9.7. In the following program, a composite data structure is constructed recursively.

```
1 function F(f) {
2   var a = this;
3   a.f = f;
4 }
```

```
5
6  F.create =
7    function (n) {
8      var f;
9      if (n < 1) {
10       f = null;
11     } else {
12       f = F.create(n-1);
13     }
14     return new F(f);
15   }
16
17 F.create(3);
```

Suppose that every object created on line 14 is allocated at a single address a. When constructor `F` on line 14 is called after the recursive call `F.create` on line 12 in the `else` branch, that recursive call has already allocated an object at address a. Therefore our purity analysis concludes that property load `a.f` on line 3 in the constructor writes to an address that already exists in the caller store. As a result, constructor `F` is considered to be impure, although a constructor should be allowed to mutate the object referenced by its `this` parameter without generating an observable side effect. □

Example 9.8. In the program from Example 9.7, we identify a second problem. Suppose that variable `f` on line 8 is always allocated at the same address. Then in a recursive call to `F.create` both assignments to `f` (lines 10 and 12) are also considered to be a write to an address that exists in the caller store. □

The solution we apply is again adding scope-based freshness for variables and objects. Variable freshness ensures that writing to a local variable does not generate an observable side effect outside the application context in which it is local. Object freshness for example ensures that mutating the newly created object in a constructor call is never considered as an observable side effect to callers.

Motivated by the evaluation of our purity analysis for Scheme (Chapter 8), we limited variable and object freshness to the topmost application context by design. Evaluation of our Scheme purity analysis indicates that few real-world benchmarks exhibit the behavior that stack-traversing scope-based freshness analysis aims to improve. This design decision enables

some simplifications in the formalization and implementation of the analysis. When the state handler of purity analysis delegates to the effect handler, it passes the topmost application context. The effect handler can then check for variable or object freshness, before initiating stack traversal using address freshness.

9.4.1 Variable Freshness

A variable is fresh with respect to a calling context if it is a local variable in that context. The handler for variable read and write effects can rely on lexical scoping information offered by predicate *isLocal* instead of checking the address of the read or written variable. Predicate $\mathit{isLocal} \subseteq \mathsf{Var} \times \mathsf{Fun}$ returns whether a variable is declared in an enclosing scope of a given function scope, or not. These new cases shortcut the previous cases of *handleEffect*.

$$\begin{aligned} \mathit{handleEffect}((\ldots,\kappa,\_),\mathbf{Wv}(a,\_),P,R,O) &= (P,R,O) \\ &\quad \text{if } \mathit{isLocal}(v,f) \\ &\quad \text{where } (\_,(f,\_),\_,\_,\_) = \kappa \end{aligned}$$

$$\begin{aligned} \mathit{handleEffect}((\ldots,\kappa,\_),\mathbf{Rv}(a,\_),P,R,O) &= (P,R,O) \\ &\quad \text{if } \mathit{isLocal}(v,f) \\ &\quad \text{where } (\_,(f,\_),\_,\_,\_) = \kappa \end{aligned}$$

9.4.2 Object Freshness

Freshness of objects referenced by variables is tracked in a map F that indicates whether a referenced object is fresh ($\bot_F$) or not (**unfr**). *Freshness* forms a join semi-lattice in which $\bot_F \sqsubset \mathbf{unfr}$, and $\bot_F \sqcup \mathbf{unfr} = \mathbf{unfr}$. Unlike object freshness analysis for SCHEME$_0$ (Section 4.4.2), we do not separately track freshness per application context. For the benchmark programs we evaluate (Section 9.6) we observed no loss in precision when applying this simplification.

$$\begin{aligned} \psi \in \mathit{Freshness} &= \{\bot_F, \mathbf{unfr}\} \\ F \in \mathit{Fresh} &= \mathsf{Var} \rightharpoonup \mathit{Freshness} \end{aligned}$$

An object reference is fresh with respect to the topmost calling context if it points to an object that was created in that context. There are two operations that need to come together: we have sources at which fresh

objects are created and/or bound to variables, and we also have to propagate object freshness.

JS_0 being a small language, sources for fresh objects are intuitive to find. The following cases define predicate $\mathit{isFresh} : \mathsf{Exp} \times \mathit{Kont} \times \mathit{Fresh}$ that captures our notion of object freshness. The application context is only required to distinguish between regular function calls and constructor calls.

1. Suppose that F is a mapping from variables to freshness of the objects they point to. Then the object a variable references is fresh if its variable is mapped onto **fr** in F.

$$\mathit{isFresh}([\![v]\!], \kappa, F) = (F(v) = \mathbf{fr})$$

2. The result of object construction through `new` is fresh.

$$\mathit{isFresh}([\![\texttt{new } v\texttt{(}s\texttt{)}]\!], \kappa, F) = \mathbf{true}$$

3. An assignment expression is fresh if its right hand side is fresh.

$$\mathit{isFresh}([\![v\texttt{=}e]\!], \kappa, F) = \mathit{isFresh}(e, \kappa, F)$$

4. A reference to the newly constructed object through `this` in a constructor is fresh.

$$\mathit{isFresh}([\![\texttt{this}]\!], ([\![\texttt{new } v\texttt{(}s\texttt{)}]\!], \ldots), F) = \mathbf{true}$$

5. All other expressions are not fresh.

$$\mathit{isFresh}(e, \kappa, F) = \mathbf{false}$$

Graph traversal

Propagation of object freshness happens through variable assignment only, following the same idea as depicted in Figure 4.1 on page 63. Resources that exist in an outer function scope with respect to some other scope are considered unfresh in the latter. Freshness is propagated through assignment within the topmost application context. As in Section 4.4.2, we do not track freshness through for example function calls or property loading and storing: for these kinds of object flow we rely entirely on the underlying abstract interpretation and address freshness. In this simplied object freshness analysis, we also do not propagate freshness of object return values.

Like purity analysis, freshness analysis piggybacks on the underlying flow graph G_e to determine control flow of a program. Function $travGraph_F$ performs graph traversal, and delegates to $handle_F$ to update freshness map F.

$$travGraph_F(S, \varnothing, F) = F$$

$$travGraph_F(S \cup \{\varsigma\}, W \uplus \{\varsigma\}, F) = F$$

$$travGraph_F(S, W \uplus \{\varsigma\}, F) = travGraph_F(S', W', F')$$
$$\text{where } W' = W \cup \{\varsigma' \mid (\varsigma \to \varsigma') \in G_e\}$$
$$F' = handle_F(\varsigma, F)$$
$$S' = \begin{cases} \varnothing \text{ if } F \neq F' \\ S \cup \{\varsigma\} \text{ else} \end{cases}$$

State handler

For a variable assignment expression, freshness propagates from the right-hand side to the variable on the left-hand side. The state handler uses predicate *isLocal* to determine whether the variable being assigned is local. If this is the case, then freshness is propagated from the right-hand side to the left-hand side. Else, the variable becomes unfresh. For any other expression, freshness remains the same.

$$handle_F(\mathbf{ev}([\![v{=}e]\!], \rho, \sigma, \iota, \kappa, \Xi), F) = F'$$
$$\text{where } F' = \begin{cases} F \sqcup [v \mapsto F(e)] \text{ if } isLocal(v, f) \\ F \sqcup [v \mapsto \mathbf{unfr}] \text{ else} \end{cases}$$

$$handle_F(\varsigma, F) = F$$

Extending Purity Analysis

We can extend purity analysis with object freshness analysis by defining handlers for property effects that shortcut purity analysis when a fresh object is involved.

A property read effect can be the consequence of an explicit property access through *lookupProp*, or when reading from a top-level variable. Because the global object is never fresh during function application, we are only interested in the former case. The machine looks up properties when

evaluating property load and method call expressions.

$$\mathit{handleEffect}(\mathbf{ev}([\![s\,.\,v]\!], \rho, \sigma, \iota, \kappa, \Xi), \mathbf{Rf}(\_, \_), P, R, O) = (P, R, O) \quad \text{if } \mathit{isFresh}(s, \kappa, F)$$

$$\mathit{handleEffect}(\mathbf{ev}([\![s_0\,.\,v\texttt{(}s_1\texttt{)}]\!], \rho, \sigma, \iota, \kappa, \Xi), \mathbf{Rf}(\_, \_), P, R, O) = (P, R, O) \quad \text{if } \mathit{isFresh}(s_0, \kappa, F)$$

Similarly, property writes can happen through explicit property store, or when assigning to a top-level variable. Again we only deal with the former case.

$$\mathit{handleEffect}(\overbrace{\mathbf{ko}(d, \sigma, \mathbf{st}(s, v, \rho) : \iota, \kappa, \Xi)}^{S}, \mathbf{Wf}(\_, \_), P, R, O) = (P, R, O) \quad \text{if } \mathit{isFresh}(s, \kappa, F)$$

Prioritizing these handlers over the property effect handlers from Section 9.3 ensures that freshness analysis for object references improves precision of the purity analysis.

9.5 Implementation

We implemented the purity analysis and freshness analysis discussed in this chapter as a proof of concept called `protopurity.js`. The prototype implementation is structured according to the approach we outlined in this chapter. At the base level, we use JIPDA[1] as the underlying abstract interpreter for producing flow graphs. Our simplified object freshness analysis runs on top of a program's flow graph. Our purity analysis also consumes this flow graph and (optionally) the results of freshness analysis that was run over the same graph.

Our implementation significantly extends JS_0 and its semantics presented at the beginning of this chapter.

Unlike the semantics formalized in this chapter, our implementation supports computed properties. Instead of working with a set of names (strings), the abstract interpreter used in our prototype works with *abstract* names that come from the value lattice used to represent all primitive values used during interpretation. Emitted read/write property events therefore also contain abstract names, and looking up property names happens through subsumption instead of equality.

[1] `https://github.com/jensnicolay/jipda`

We also added many of the built-in JavaScript functions and objects required to run our benchmarks.

Our prototype implementation uses abstract garbage collection [Might and Shivers, 2006a] as a technique to increase performance and precision of abstract interpretation (Section 3.5). Abstract garbage collection reclaims unused addresses, and increases the precision of an address-based purity analysis of our Scheme procedure side-effect analysis (Section 8.5.3). For our JavaScript purity analysis, however, the advantage of abstract garbage collection in terms of precision was less pronounced. Disabling abstract garbage collection on smaller and synthetic benchmarks only lead to a small negative impact on precision, which was dominated by the absence or presence of freshness analysis. However, abstract garbage collection was required to run the larger benchmarks in a reasonable amount of time and space. Scaling up the abstract interpreter and client analyses, also to better assess the impact of garbage collection for `protopurity.js`, is future work.

Finally, we mention that our abstract machine implements abstract counting [Might and Shivers, 2006a]. Abstract counting is a technique that keeps track of whether an address is allocated exactly once or multiple times in the store. If the address has only been allocated once, a strong update can be used instead of a weak update (Section 2.4.3). When running our JavaScript benchmarks, we found that abstract counting boosted precision considerably. JavaScript hoists variable declarations to the top of their defining function scope, and our abstract machine semantics initializes these variables to `undefined`. Only afterwards is the variable *assigned* its "initial" (from the perspective of the program) value. Because of abstract counting, this happens using a strong update, thereby avoiding the precision loss incurred by merging.

9.6 Experiments

We used our prototype implementation `protopurity.js` to analyze several JavaScript programs using 0CFA allocation (Section 9.2.2).

We distinguish between two types of test programs: small unit test programs that test a specific feature which is challenging for purity analysis, and larger existing benchmark programs. We report on the correctness, soundness, precision, and performance of purity analysis results (Section 8.1.2) on these programs. In contrast to the experiments with `purity.rkt` for

		— Without fresh —			— With fresh —		
Benchmark	Functions	**pure**	**obs**	**proc**	**pure**	**obs**	**proc**
`access-nbody`	11	1	1	9	6	1	4
`controlflow-recursive`	3	3	0	0	3	0	0
`crypto-sha1`	8	6	0	2	7	0	1
`math-spectral-norm`	5	2	0	3	2	0	3
`tree-add`	6	1	0	5	3	1	2
`navier-stokes`	29	3	1	25	3	1	25
`richards`	32	4	1	27	8	1	23
`bisort`	13	4	1	8	5	1	7
`em3d`	13	3	0	10	4	0	9
`mst`	18	5	1	12	7	1	10

Table 9.1: Purity analysis results. For each benchmark, we report the number of functions in each effect class as determined by our analysis. We ran the analysis without and with using the results from scope-based freshness analysis. *Functions* is the total number of functions called in the benchmark.

SCHEME$_0$ (Chapter 8), we focus more on the precision of analysis results, and on a comparison with existing techniques.

9.6.1 Correctness and Soundness

We manually specified the expected function classification (**pure**, **observer**, or **procedure**) for an extensive set of unit tests and benchmark programs. We mechanically checked that the actual results of concrete purity analysis are equal to the expected outcome. From this evaluation we conclude that, for the set of programs under test, our implementation is correct.

We also checked that the classification results of abstract purity analysis subsume the results of the concrete analysis. From this we conclude that the results for our set of test programs are sound.

9.6.2 Precision

We manually verified the result of function classification by our purity analysis in a large set of programs, and found few false positives (i.e., functions reported as pure by concrete purity analysis but not by abstract purity analysis).

Table 9.1 reports, for a selected set of 10 existing benchmark programs, the number of pure functions detected, for both our purity analysis without

Benchmark	Flow time	— Without fresh — Purity time	— With fresh — Purity time
`access-nbody`	ε	ε	ε
`controlflow-recursive`	ε	ε	ε
`crypto-sha1`	ε	ε	ε
`math-spectral-norm`	ε	ε	ε
`tree-add`	0'01"	ε	ε
`navier-stokes`	2'50"	3'18"	3'08"
`richards`	4'33"	0'50"	0'37"
`bisort`	0'07"	0'03"	0'01"
`em3d`	0'02"	0'12"	0'10"
`mst`	0'04"	0'15"	0'12"

Table 9.2: Flow and purity analysis timing. *Flow time* is the running time of the flow analysis creating a flow graph. *Purity time* is the running time of the purity analysis on top of the flow graph. We use ε to denote a running time smaller than 1 second.

scope-based freshness analysis, and for our purity analysis combined with scope-based freshness analysis.

`crypto-sha1` is a Sunspider benchmark[2] that tests cryptographic functions. Our abstract purity analysis was able to detect that the majority of the functions that are actually called in that benchmark are indeed pure functions.

`navier-stokes` is an Octane benchmark[3] that passes around arrays between functions that update these arrays in place. Our abstract analysis correctly predicted that almost all functions in this benchmark are impure.

Extrapolating from these results, we conjecture that `protopurity.js` is capable of determining function purity with sufficient precision to be useful to client applications.

Impact of scope-based freshness

From the benchmark results we see that incorporating scope-based variable and object freshness analysis improves precision significantly.

`tree-add` is a JOlden benchmark [jol] that we converted from Java into JavaScript. Although it is a relatively small benchmark, it exhibited poor precision when analyzed without variable and object freshness: only 1 pure

[2] `https://www.webkit.org/perf/sunspider/sunspider.html`
[3] `https://developers.google.com/octane`

			ReImInfer			JPPA			JPure		
Program	#Fun	pp.js	=	+	−	=	+	−	=	+	−
bisort	11	5	5	1	1	6	1	0	5	1	1
tree-add	6	3	3	0	0	3	0	0	2	0	1
em3d	14	6	5*	0	1	4	0	2	4	0	2
mst	18	6	5	0	1	5	0	1	5	0	1

Table 9.3: Comparison between multiple purity analyses on benchmarks from the JOlden suite. *#Fun* is the number of functions in the Javascript benchmark. Column `pp.js` shows the number of functions identified as pure (**pure** or **observer** in our terminology) by `protopurity.js`. *RemImInfer*, *JPPA* and *JPure* show the number of functions detected as pure by the other tools: = counts the methods which are detected as pure by both `protopurity.js` and the other tool, + shows the number of functions detected as pure by the other tool but not by `protopurity.js`, and − shows the number of functions identified as pure by `protopurity.js` but not by the other tool. An asterisk indicates that a tool incorrectly reported an impure function as pure.

function was detected out of 4 functions that are determined pure with scope-based freshness analysis enabled. The pattern in Example 9.7, illustrating some of the weaknesses of address-only purity analysis, was distilled from this benchmark.

9.6.3 Performance

We also report in Table 9.2 the time the underlying flow analysis and the subsequent purity analysis took, again without and with scope-based freshness analysis.

Table 9.2 lists, for our set of 10 benchmark programs, the running time of flow analysis, and the running time of abstract purity analysis with and without scope-based freshness.

We observe that incorporating scope-based freshness analysis improves the overall running time of the analysis. This is because scope-based freshness in `protopurity.js` avoids stack traversal when effects occur on fresh resources.

Comparison to Existing Work

Comparing our approach to existing approaches in terms of results is difficult. To the best of our knowledge, our previous work [Nicolay et al., 2015] is the first purity analysis that specifically targets JavaScript. The analysis we

present here extends our original analysis into one that additionally handles read effects and is therefore capable of classifying functions into a ternary scheme instead of a binary one. Other related work, which we discussed in Section 6.5, focuses primarily on method purity for Java, and analyzes benchmarks from the JOlden suite.

Programs `tree-add`, `bisort`, `em3d`, and `mst` are JOlden benchmarks that we manually converted to JavaScript. We compare our results (`protopurity.js`) for these benchmarks with ReImInfer [Huang et al., 2012], JPPA [Salcianu and Rinard, 2005] and JPure [Pearce, 2011] in Table 9.3. We manually verified and compared the results reported by each tool on every method or function that is present in both the original program and the JavaScript translation of the benchmark. In the `bisort` benchmark, one pure function is not detected by `protopurity.js`, but is detected by the other tools. For the `tree-add`, `em3d` and `mst` benchmarks, `protopurity.js` detects every pure function as pure, while none of the other tools detect every pure function. One spurious result was reported by *ReImInfer* on the `em3d` benchmark, and is not included in the listed counts.

From this experiment we conclude that for these benchmarks our tool is as precise as existing approaches, and often is able to identify more pure functions.

9.7 Related Work

To the best of our knowledge, our previous work [Nicolay et al., 2015] is the first purity analysis that specifically targets JavaScript.

We use JIPDA[4] as the underlying abstract interpreter for producing flow graphs. There exist several other JavaScript analysis tools such as WALA [wal], TAJS [Jensen et al., 2009], and JSAI [Kashyap et al., 2014]. We conjecture that it is possible to implement our approach in these tools. JIPDA was constructed as a prototype for analyzing a small subset of JavaScript following the AAC method [Johnson and Van Horn, 2014] based on abstract machines, and therefore exposes the program representation and information required for performing our purity analysis by design. Enabling the implementation of our approach on top of existing tools therefore necessitates the effort of extending or adapting these tools so that the required program information becomes available.

[4] `https://github.com/jensnicolay/jipda`

Any implementation of our side-effect and purity analysis requires a flow analysis that models store-allocated resources, effects on these resources, and the call stack. It must also be able to associate the applied procedures and caller stores with each application. Additionally, our object freshness and purity analysis require a flow-sensitive program analysis. We formulated our analyses on top of a flow graph that exposes the necessary information—more specifically nodes in this graph represent program states and edges are annotated with effects that occur on transition between states.

9.8 Conclusion

We presented a purity analysis for JS_0 that handles closures, higher-order functions, and prototypal inheritance. The primary goal was to investigate if our side-effect and purity analysis for SCHEME_0 could be adapted to a more dynamic and complex setting, and how it would perform. We were also interested in evaluating the effect of simplifying some of the analyses on precision and performance.

The design of the analysis follows the same outline as the one we developed for Scheme in previous chapters. However, in the setting of JS_0, the analysis has to deal with things like prototypal inheritance, computed properties, constructor functions, and implicit updates of object properties. We also added scope-based freshness analysis to our purity analysis, but restricted the freshness checks to the topmost application context. This decision simplified the design of scope-based freshness analysis significantly, but not at the cost of performance and precision. Adding scope-based freshness analysis improves precision of our purity analysis considerably, and reduces execution time, as demonstrated in our experiments. When comparing our purity analysis for JavaScript against existing solutions, we find that our implementation is as precise or more precise than results produced by other tools.

Chapter 10

Conclusion

10.1 Summary

Although side effects are essential to many programs, they encumber understanding a program's behavior. Therefore, determining side effects, or the absence thereof, is useful for software engineering purposes and program optimization. It is, however, also a difficult problem, and particularly so in the presence of compound data values, objects, and higher-order procedures.

This dissertation presented and evaluated techniques for statically computing the side-effecting behavior of higher-order imperative programs to determine procedure purity. Pure procedures do not generate observable side effects and, depending on the definition of purity used, may or may not depend on external side effects.

In practice it is not sufficient to merely detect the presence of side-effecting expressions in a program. The dynamic extent of side effects must also be established accurately to be useful for performing purity analysis. In this dissertation, we used stack reachability and resource freshness to determine the extent of side effects.

The principle of stack reachability [Might and Prabhu, 2009], transposed to our side-effect analysis, states that a side effect occurs for all procedures on the call stack. Every procedure call generates an application context, and the stack represents all application contexts, from the most recent on top of the stack to the root context at the bottom. A single side effect may be observable by some callers and unobservable by others.

To determine observability with respect to application contexts, we used the concept of freshness. If a resource is created in the application context where it is used, then that resource is not reachable from callers, and we

regard these resource as fresh. Effects on fresh resources are unobservable by callers and can be masked. We identified and formalized three characterizations of freshness.

- *Address freshness* When an address is not in the caller store of an application context, then that address is fresh with respect to that context.
- *Variable freshness* A variable is fresh with respect to an application context if the variable is local to the procedure scope of the applied procedure.
- *Object freshness* An object is fresh with respect to an application context if the object was created in that application context.

In order to compute the program properties required for determining stack reachability and resource freshness, we based ourselves on the Abstracting Abstract Control approach [Johnson and Van Horn, 2014]. We defined abstract state machines that are interpreters for core Scheme and JavaScript languages, and which are instrumented to register read and write effects on variables and objects. These machines are abstracted using abstract interpretation, and parameterized to be able to express both concrete and abstract semantics. The result of program interpretation is a flow graph that can be consumed by client analyses which are interested in program properties involving control flow, value flow, and effects.

We designed our purity analysis as a number of different analyses that consume each other's results, starting from a flow analysis. We applied these analyses to a set of 13 benchmark programs, and discuss the outcome in terms of correctness, soundness, precision, and performance. Our experiments show that the purity analysis is capable of uncovering purity in a variety of programs. Purity analysis is able to correctly classify procedures as **pure**, **observer**, or **procedure**, based on the side effects they generate and depend upon. In larger benchmarks, the precision for detecting observers in larger benchmarks is sometimes low, and extending the analysis with scope-based freshness does not significantly improve precision. Despite these limitations, we find that in general the purity analysis detects side effects and pure procedures with a precision that enables applications in software engineering. The purity analysis is precise in distinguishing procedures that generate observable side effects (**procedure**) from those that do not (**pure**, **observer**). Incorporating scope-based variable and object freshness and escape analysis generally yields significant improvements in precision.

10.2 Restating The Contributions

We restate the main contributions this dissertation makes.

Our first contribution is a procedure side-effect analysis that computes for each procedure application the side effects that are observable by direct and indirect callers. Applications and associated callers are found by traversing all reachable application contexts on the call stack at the point where an effect occurs. Observability of effects is based on freshness of resources, and we initially formulate an analysis based on address freshness.

However, address freshness does not always offer the best precision because termination of static analysis is primarily guaranteed by allocating resources at addresses that are already in use, and therefore not fresh. Our second contribution is the introduction and formalization of two additional scope-based characterizations of freshness to improve precision: variable freshness and object freshness, in combination with escape analysis. Variable freshness is based on locality of variables, i.e., whether they are local or free with respect to a procedure's scope. Object freshness tracks the flow of objects in and out of scopes through object references to determine in which application contexts the referenced objects are fresh.

Our final contribution is the design of a purity analysis on top of procedure side-effect analysis. Purity analysis summarizes how each procedure behaves during program execution in terms of side effects during its applications. It considers two types of side-effecting behavior: generating observable side effects, and depending on external side effects. The result of purity analysis is a mapping from procedures to their side-effecting behavior, which we use to further classify procedures by mapping effect summaries to different definitions of purity that exist in the literature. The most precise classification distinguishes between pure procedures, observers, and all other procedures. A procedure is pure if none of its applications generate or depend upon externally observable side effects. A procedure is an observer as soon as one of its applications depends on an external side effect, but none of its applications generate observable side effects. Otherwise, a procedure is classified as simply a procedure.

In summary, this dissertation makes the following contributions.

- A *procedure side-effect analysis* that computes observable side effects for each application context reachable on the stack (Chapter 3).
- A *freshness analysis for addresses, variables, and objects* (Chapter 4), extended with escape analysis for increased precision (Chapter 5).

- A *purity analysis* for higher-order, imperative programs (Chapters 6 and 9).

10.3 Limitations and Future Work

Despite the fact that our purity analyses for SCHEME$_0$ and JS$_0$ are capable of determining procedure purity with sufficient precision to be useful in a number of software engineering scenarios, there is still room for improvement. We identified the following limitations, primarily concerning the soundness and precision of our approach. Overcoming these limitations is what we consider to be future work.

10.3.1 Soundness of Analyses

We empirically demonstrated that the results produced by our implementations in Chapter 8 and Section 9.6 are sound for a large suite of challenging benchmark programs. Empirical soundness was established by verifying that every abstract analysis result subsumes the corresponding concrete result. We believe that this enables the use of our implementation in a variety of scenarios that require sufficient confidence in the soundness of the results. Using the analysis in an optimizing compiler, however, would require a more thorough soundness proof.

Assuming the flow analysis is sound, we believe that formally proving soundness of address-based purity analysis (Section 3.4) would present no major obstacles. This is because effects and address freshness represent minor additions to the original abstract state-space and semantics of the flow analysis, respectively.

Proving correctness of concrete scope-based freshness, especially the object freshness analysis for SCHEME$_0$ in its current form (Section 4.4.2), and proving it sound in the abstract semantics, is more challenging future work.

10.3.2 Precision of Flow Analysis

Evaluation of the implementation of our flow analysis demonstrated low precision on large benchmark programs that are branch-heavy and contain many allocation sites. The negative consequences of diminished precision in both control flow and object allocation amplify each other, resulting in a graph that has low overall precision.

Improvements in the precision of the flow analysis would therefore benefit the purity analysis. Possibilities for achieving this include more elaborate resource allocation schemes (e.g., different forms of context-sensitivity [Milanova et al., 2002]), the application of heuristics during flow analysis, and the development of additional supporting analyses.

10.3.3 Definition of Object Freshness Analysis

Object freshness analysis (Section 4.4.2) was designed as a supporting analysis for procedure side-effect analysis. However, the analysis for SCHEME_0 attaches variables to application contexts. Therefore its mechanisms for variable lookup and update are necessarily imprecise. This is because variables, unlike addresses, do not necessarily represent the same resources in different application contexts on the stack. For soundness reasons, object freshness analysis uses weak updating for freshness, even in concrete semantics. Therefore, object freshness analysis does not offer full precision, even when the underlying flow graph does.

Compared to the object freshness analysis for SCHEME_0, the analysis for JS_0 was simplified by no longer tracking object freshness per application context (Section 9.4). Evaluation of this approach for JS_0 (Section 9.6) shows that this simplified object freshness analysis still improves the precision and also improves the performance of purity analysis.

As future work, we propose to either look into simplifying object freshness analysis for SCHEME_0 in the same way as it was simplified for JS_0, or, on the contrary, make object freshness analysis more precise. Inspiration for improving object freshness analysis can come from existing flow and escape analyses that identify or handle stack allocation of variables and escaping of references. CFA2 [Vardoulakis and Shivers, 2010], for example, is a pushdown flow analysis for higher-order languages that differentiates between stack and heap references. The stack in CFA2 is therefore not only a control structure, but also contains variable bindings. Stack references are looked up in the top frame, and heap references in the store.

Modifying our object freshness analysis (or the underlying flow analysis) so that freshness analysis becomes fully precise in the concrete, and evaluating the impact on precision in the abstract, is an interesting but challenging avenue for future work.

10.3.4 Precision of Detecting Observers

The evaluation of our approach made it clear that purity analysis has difficulty distinguishing pure procedures from observers in larger programs. Observers are procedures of which at least one application depends on an external side effect, but none of its applications generate observable side effects.

Observers are identified by detecting read–write–read sequences on a particular resource, in which the first and last read must happen by the same procedure (i.e., the observer). This approach was chosen because in the concrete, in which flow and effect information has full precision, that particular sequence provides a close approximation of procedures for which return values depend on resources other than their direct parameters.

Purity analysis on its own represents the largest part of the overall running time. Detecting observable write effects is very fast because it does not involve the detection of a sequence: one observable write effect suffices for a procedure to be marked as **procedure**. However, detecting observers is expensive in terms of running time, and does not result in high precision. It would therefore be interesting to investigate other ways of detecting observers, possibly with different tradeoffs between speed and precision.

10.4 Future Research

In this section, we discuss some future avenues of research, aimed at improving the applicability of our approach.

10.4.1 Memoization

Memoization is a fundamental optimization technique that avoids recomputing previously computed results by storing them in a table from which they can be looked up, effectively trading in space for time [Acar et al., 2003]. In the context of programming languages, a typical memoization target is a procedure call. If calling conditions match previously seen calling conditions, then execution of the procedure body is skipped and cached values are returned instead. If memoization is done at the program level (i.e., not in the interpreter) and only returned values are cached (but no other program state), then the memoized procedure must be pure. Although the goal of memoization is to increase performance, memoization itself introduces

overhead because it needs to store cached results and requires lookups in this cache.

A memoization analysis would build upon purity analysis by requiring procedures that are candidates for memoization to be pure (Chapter 6) in the strictest sense. Unlike the purity analysis presented in this work, memoization also has to take fresh resources as return values of procedures into account. For example, if a procedure returns a fresh pair, and that pair is cached in the memo table, then the pair is shared between all callers of the memoized procedure. This is problematic if the pair is subsequently modified. Our freshness analysis (Chapter 4) could detect this situation. An additional verification on procedures that do return fresh resources, would be to disallow subsequent modification of the returned resource. This would ensure that resources cached as return values are immutable.

Another difficulty for memoization analysis is the common scenario in which resources are passed as arguments to procedures. Memoization has to decide under which condition resources are equal. For example, if a pair is passed as an argument, the memoization algorithm has to decide during memo lookup whether the argument pair is equivalent to a cached entry. This can happen by comparing memory locations (`eq?`) or by recursively comparing the contents of the pair (`eqv?` or `equal?`). We agree with Finifter et al. [2008] that equivalence of arguments is a parameter of procedure memoization.

10.4.2 Referential Transparency

Expressions that are referentially transparent can always be replaced by their value without changing the behavior of the program [Søndergaard and Sestoft, 1990]. Referentially transparent expressions must be pure, but additionally should not allocate resources.

Our approach can be extended to including tracking of allocation effects, so that referential transparency can become a classification of procedures in our purity analysis that is "above" purity: all referentially transparent procedures are pure, but the inverse is not necessarily true.

10.4.3 Parallelization

In Nicolay et al. [2011], we describe a program transformation that automatically parallelizes higher-order imperative Scheme programs. The approach first transforms the input program into administrative normal form. This

serves two purposes: it introduces many series of nested `let` expressions that are the target of the automatic parallelization approach, and it renders the program in a form that is suitable for dependence analysis by imposing an order of evaluation.

The key idea is to look at every series of nested `let` expressions that appear in a program, and decide where it is possible to evaluate binding expressions in parallel. The approach must be instantiated with a sound and sufficiently precise dependence analysis, for dependent expressions cannot safely be evaluated in parallel. Expression e_2 is *dependent* on expression e_1 if e_1 is evaluated before e_2 and e_2 accesses or modifies a resource that was accessed or modified before by e_1.

To generate parallel code, we construct a *binding dependency graph* that models the temporal evaluation constraints imposed by the dependencies on variable binding in a program. This binding dependency graph is a directed, acyclic graph (DAG) in which each node represents a single variable binding operation in a nested `let`.

The implementation of the underlying flow and dependence analysis in Nicolay et al. [2011] has some limitations. It builds on the finite-state dependence analysis by Might and Prabhu [2009]. It also does not separately model the `car` and `cdr` fields of a pair, but considers pairs to be a single resource, which leads to a precision penalty. Similarly, it does not model the individual elements of a vector.

The approach presented in this dissertation overcomes these shortcomings. We use a pushdown analysis that offers more precise call/return precision than a classic finite-state analysis. Our side-effect analysis models the individual fields of pairs, and the individual elements of a vector (up to the precision offered by the value lattice). We are confident that the side-effect analysis we presented in this work is capable of assuming the role of dependence analysis in techniques for parallelizing Scheme programs.

10.5 Concluding Remarks

We started out this dissertation with the observation that popular imperative programming languages have adopted higher-order programming, a feature originating in the functional programming paradigm. Programs written in these higher-order imperative languages gain some of the same benefits from higher-order programming as functional languages, such as increased expressivity and modularity. At the same time, these programs still contain

side effects, rendering program comprehension and automated verification more difficult.

To enable higher-order imperative programming to more fully exploit the advantages of functional programming, we presented a static analysis for determining side effects and their dynamic extent as precisely as possible, with the goal of detecting procedure purity.

This dissertation demonstrates that procedure purity can be effectively approximated statically in higher-order, imperative programs by using an abstract machines approach.

We implemented a proof-of-concept of our purity analysis for relevant subsets of both Scheme and JavaScript. Evaluation of our implementation shows that purity analysis produces correct, sound, and precise results for a large set of benchmarks.

We also identified a number of limitations in our approach, especially concerning lower precision in larger programs containing many allocation sites and conditionals.

Despite the limitations, we believe our side-effect and purity analyses are a useful contribution to, and a promising foundation for applications in software engineering.

Appendix A

Notation and Conventions

Disjoint union We use $\uplus$ to denote disjoint union. If $X = Y \uplus Z$, then $Y = X \setminus Z$.

Sequences The notation $X = x : X'$ deconstructs a sequence X into its first element x and the rest X'. We write $\langle\rangle$ for the empty sequence.

Power domain The power domain of set X is denoted as $\mathcal{P}(X)$.

Empty function The empty function is denoted as $[]$, and for all inputs returns the bottom element $\bot$ of its range.

Function extension The notation $f[x \mapsto y]$ yields a function f' such that:

$$f'(z) = \begin{cases} y & \text{if } z = x, \\ f(z) & \text{else.} \end{cases}$$

Function restriction We write the restriction (or narrowing) of a function f to domain X as $f|X$, such that $(f|X)(x) = f(x)$ if $x \in X$ and $(f|X)(x) = \bot$ else.

Function joining Joining of functions happens in a pointwise fashion. If $\sqcup$ is the join operator for the range of the function, then $[x \mapsto y_1] \sqcup [x \mapsto y_2] = [x \mapsto y_1 \sqcup y_2]$. In particular, $\bigsqcup\{[x_0 \mapsto y_0], \ldots, [x_n \mapsto y_n]\} = [x_0 \mapsto y_0] \sqcup \ldots \sqcup [x_n \mapsto y_n]$.

Function cases When a function is defined using several (numbered) cases, then it is assumed that the cases are considered in order and do not fall through.

Appendix B

Abstractions, Lattices, and Fixpoints

This dissertation presents an approach for detecting side effects in higher-order programs that is based on an abstract interpretation of those programs. We present an overview of the necessary background in this appendix.

B.1 Abstractions and Concretizations

Abstraction is the single most important tool of a programmer, and therefore every programming language essentially is a collection of abstraction mechanisms at the disposal of the programmer. Abstraction allows one to focus on one part of a problem while abstracting away other details.

Value abstractions are often encountered in computing [Schmidt, 2003]. An abstract value "names" or represents one or more concrete values. In what follows, we denote abstract values with angle brackets that delimit a set of concrete values (e.g., $\langle 2, 5\rangle$ represents the set of concrete values 2 and 5), or as a named set (e.g., Int), sometimes with a "hat" (e.g., $\widehat{Int}$) to stress the fact that it is the abstract counterpart of the concrete set under the hat.

Example B.1. If $x \sqsubset y$ denotes that y is less precise than x, then $\langle 2\rangle \sqsubset \langle 2, 5\rangle \sqsubset \langle 0..9\rangle \sqsubset \widehat{Int}$ are all abstractions of the number 2. □

An abstraction function α maps a set of concrete values to an abstract value that offers the most precise *approximation*.

Example B.2. If the set of abstract values for number 2 from Example B.1 are the only abstract values available, then $\alpha(\{2\}) = \langle 2\rangle$ and $\alpha(\{1, 2, 3, 42\}) = \widehat{Int}$. □

Conversely, the concretization function γ maps an abstract value to the set of concrete values it represents.

Example B.3. $\gamma(\langle 2,5\rangle) = \{2,5\}$ and $\gamma(\widehat{Int}) = \{\ldots,-1,0,1,\ldots\}$. □

When dealing with approximations, the abstraction mapping α introduces *imprecision*, as the following example illustrates.

Example B.4. We can have $\alpha(\{2,5,6\}) = \langle 0..9\rangle$ but $\gamma(\langle 0..9\rangle) = \{0,1,\ldots,9\}$, so that $(\gamma \circ \alpha)(\{2,5,6\}) \neq \{2,5,6\}$. □

Because $\{2,5,6\} \subseteq \{0,1,\ldots,9\}$ we say that α is *sound*.

While $(\gamma \circ \alpha)$ is inexact, it is easy to verify that the inverse $(\alpha \circ \gamma)$ never loses precision.

More formally, if for a set of concrete values X, with $\mathcal{P}(X)$ denoting the power set of X, and its abstract counterpart $\hat{X}$ we have:

$$\forall c \in \mathcal{P}(X), a \in \hat{X} : c \subseteq \gamma(a) \iff \alpha(c) \sqsubseteq a$$

then the abstraction is sound. The above relation between X and $\hat{X}$ is called a *Galois connection*.

Abstractions are not limited to values but also extend to operations. Suppose we have a concrete operation f defined on concrete domain X. $\hat{f}$ is a sound abstraction of f if the following holds [Cousot and Cousot, 1992]:

$$\forall x \in X : f(x) \sqsubseteq \gamma(\hat{f}(\alpha(x)))$$

or, equivalently,

$$\forall \hat{x} \in \hat{X} : \alpha(f(\gamma(\hat{x}))) \sqsubseteq \hat{f}(\hat{x})$$

Example B.5. (Sign abstraction) A widely used example is the abstraction of integers to their sign [Cousot and Cousot, 1977, Might and Shivers, 2008]. The concrete values are the integers $\mathbb{Z}$, while the abstract values are in the power set of signs $\hat{\mathbb{Z}} = \mathcal{P}(\{-,0,+\})$. For every $z \in \mathbb{Z}$ we have

$$\alpha(z) = \begin{cases} \langle - \rangle & z < 0 \\ \langle 0 \rangle & z = 0 \\ \langle + \rangle & z > 0 \end{cases}$$

The addition operator $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ abstracts naturally to $\oplus : \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}}$ following the rule of signs, so that for example:

$$\begin{aligned} \langle 0 \rangle \oplus \langle 0, + \rangle &= \langle 0, + \rangle \\ \langle + \rangle \oplus \langle + \rangle &= \langle + \rangle \\ \langle + \rangle \oplus \langle - \rangle &= \langle -, 0, + \rangle \\ \langle +, - \rangle \oplus \langle 0 \rangle &= \langle -, + \rangle \end{aligned}$$

The abstract execution of $4 + (-4)$ yields $\langle -, 0, + \rangle$, while concrete execution of $4 + (-4)$ and then abstracting would yield $\alpha(4 + (-4)) = \langle 0 \rangle$, showing that abstract interpretation strictly over-approximates. Furthermore, $\{4 + (-4)\} = \{0\} \subseteq \{-, 0, +\} = \gamma(\langle + \rangle \oplus \langle - \rangle)$ supports the fact that $\oplus$ is a sound abstraction of $+$. □

B.2 Lattices and Fixpoints

Our example of abstracting the number 2 in Appendix B.1 uses a very simple ordered set of abstract values. We now give a mathematical characterization of the sets involved in abstract interpretation, following Schmidt [2003], Schwartzbach [2008].

A *partial order* is a binary relation $\sqsubseteq$ over a set P such that for all a, b, and c in P:

- $a \sqsubseteq a$ [reflexive]
- if $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$ [antisymmetric]
- if $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq b$ [transitive]

$(P, \sqsubseteq)$ is a *partially ordered set* (or *poset*) if P is a set with partial order $\sqsubseteq$.

Let A be a subset of poset $(P, \sqsubseteq)$. An element p in P is an *upper bound* for A if $a \sqsubseteq p$ for every a in A. If p is the smallest among the upper bounds, then p is the *least upper bound* or *supremum* of A ($\sup A$). An element p in P is a *lower bound* for A if $p \sqsubseteq a$ for every a in A. If p is the greatest among the lower bounds, then p is the *greatest lower bound* or *infimum* of A ($\inf A$).

A *join semilattice* is a poset in which any two elements a and b have a unique least upper bound or *join*, denoted $a \sqcup b$. A *meet semilattice* is a poset in which any two elements a and b have a unique greatest lower bound or *meet*, denoted $a \sqcap b$. A poset is a *lattice* if it is both a join semilattice

and meet semilattice. A lattice $(L, \sqsubseteq)$ is *bounded* if there exists a maximum element (*top* or $\top$) and a minimum element (*bottom* or $\bot$) in L such that for every x in L:

- $x \sqsubseteq \top$
- $\bot \sqsubseteq x$

In a bounded lattice, $\bot$ is the identity element for $\sqcup$, and $\top$ is the identity element for $\sqcap$.

A lattice $(L, \sqsubseteq)$ is *complete* if every subset A of L has both a join (denoted $\sqcup A$) and a meet (denoted $\sqcap A$) in L. Any finite lattice is trivially a complete lattice. If $(L, \sqsubseteq)$ is a complete lattice, then it is a bounded lattice with:

- $\top = \sqcup L$
- $\bot = \sqcap L$

Example B.6. The powerset lattice $(\mathcal{P}(X), \subseteq)$ of a set X, with join $\cup$ and meet $\cap$, is a complete lattice with infimum $\varnothing$ and supremum X. □

A function $f : L \to L$ is *order-preserving* or *monotone* if for all x and y in X we have:

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y)$$

In a complete lattice $(L, \sqsubseteq)$, every monotone function $f : L \to L$ has a unique *least fixpoint*, denoted $\text{lfp}(f)$, such that:

$$\text{lfp}(f) = \sup(\{f^n(\bot) \mid n \in \mathbb{N}\})$$

This final result is called the Kleene fixpoint theorem.

Appendix C

Tables

This appendix contains additional tables with results from the evaluation of `purity.rkt`. These tables represent the numerical data on which the bar charts in Chapter 8 are based.

Benchmark	Obs A	Obs VA	Obs OVA	Obs EOVA
`fib`	14	6	6	6
`purity65`	5	2	1	1
`purity73`	22	17	17	14
`purity75`	27	12	9	9
`treeadd`	116	48	45	45
`nqueens`	125	89	89	89
`dderiv`	1154	901	901	901
`destruc`	214	194	194	194
`grid`	241	190	190	184
`matrix`	151	145	145	142
`fannkuch`	260	239	239	238
`mceval`	32752	32108	32108	32091
`scm2java`	21129	20713	20713	20713

Table C.1: Impact of scope-based freshness analysis and escape analysis on the precision of procedure side-effect analysis. *Obs* represents the number of observable side-effects for all effects in all application contexts, as computed by procedure side-effect analysis.

	— A —		— EOVA —	
Benchmark	No GC	GC	No GC	GC
`fib`	15	14	6	6
`purity65`	6	5	1	1
`purity73`	38	22	14	14
`purity75`	43	27	9	9
`treeadd`	141	116	45	45
`nqueens`	285	125	113	89
`dderiv`	1621	1154	901	901
`destruc`	596	214	229	194
`grid`	297	241	184	184
`matrix`	484	151	160	142
`fannkuch`	711	260	262	238
`mceval`	36230	32752	32939	32091
`scm2java`	24596	21129	20811	20713

Table C.2: Impact of abstract garbage collection on the precision of procedure side-effect analysis using two abstract combinations A and EOVA. *No GC* represents the number of observable side-effects for all effects in all application contexts without abstract garbage collection when determining address freshness; *GC* represents the same number but with abstract garbage collection enabled.

Bibliography

CHICKEN Scheme: A practical and Portable Scheme System. URL http://www.call-cc.org.

JOlden Benchmarks. URL ftp://ftp.cs.umass.edu/pub/osl/benchmarks.

T.J. Watson Libraries for Analysis (WALA). URL http://wala.sf.net.

Harold Abelson and Gerald Jay Sussman. Structure and interpretation of computer programs. 1983.

Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, Norman I Adams IV, Daniel P. Friedman, E Kohlbecker, Guy L Steele Jr, David H Bartley, Robert Halstead, et al. Revised5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

Umut A Acar, Guy E Blelloch, and Robert Harper. *Selective memoization*, volume 38. ACM, 2003.

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. *Orbit: An optimizing compiler for Scheme*, volume 21. ACM, 1986.

Kenneth R. Anderson and Duane Rettig. Performing lisp analysis of the fannkuch benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, October 1994. ISSN 1045-3563.

John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

Anindya Banerjee and David A Schmidt. Stackability in the simply-typed call-by-value lambda calculus. In *Static Analysis*, pages 131–146. Springer, 1994.

Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. Allowing state changes in specifications. In *Emerging Trends in Information and Communication Security*, pages 321–336. Springer, 2006.

Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *Applied Semantics*, pages 42–122. Springer, 2002.

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for java. *Acm Sigplan Notices*, 34(10): 1–19, 1999.

John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):1029–1052, 2004.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP 2007–Object-Oriented Programming*, pages 28–53. Springer, 2007.

Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *Proceedings of the 2010 Workshop on Scheme and Functional Programming (Scheme 2010)*, Montreal, Quebec, Canada, August 2010.

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. *ACM SIGPLAN Notices*, 47(9):177–188, 2012.

Mattias Felleisen and Daniel P Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 314–. ACM, 1987.

Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in Java. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174. ACM, 2008.

C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993.

Matthew Flatt. Plt. reference: Racket. Technical report, Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang. org/tr1, 2010.

Brent Fulgham and Isaac Gouy. The computer language benchmarks game. *http://shootout. alioth. debian. org*, 2009.

Richard P Gabriel. *Performance and evaluation of LISP systems*, volume 263. MIT press Cambridge, Mass., 1985.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43th Annual ACM Symposium on the Principles of Programming Languages (POPL 2016)*, St. Petersburgh, Florida, USA, January 2016.

III Harrison, Williams Ludwell. The interprocedural analysis and automatic parallelization of scheme programs. *LISP and Symbolic Computation*, 2 (3-4):179–396, 1989.

Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. Reim & reiminfer: Checking and inference of reference immutability and method purity. In *ACM SIGPLAN Notices*, volume 47, pages 879–896. ACM, 2012.

John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55. Springer, 2011.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

James Ian Johnson and David Van Horn. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 11–22. ACM, 2014.

Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: a static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

John M Lucassen and David K Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM, 1988.

Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani. Purity analysis: An abstract interpretation formulation. In *Static Analysis*, pages 7–24. Springer, 2011.

Stefan Marr, Jens Nicolay, Tom Van Cutsem, and Theo D'Hondt. Modularity and conventions for maintainable concurrent language implementations: A review of our experiences and practices. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, pages 21–26. ACM, 2012.

Matthew Might and Panagiotis Manolios. *A posteriori* soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009)*, Savannah, Georgia, USA, January 2009.

Matthew Might and Tarun Prabhu. Interprocedural dependence analysis of higher-order programs via stack reachability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming (Scheme 2009)*, Boston, Massachussetts, USA, August 2009.

Matthew Might and Olin Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, volume 41, pages 13–25. ACM, 2006a.

Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*, pages 13–25, Portland, Oregon, September 2006b.

Matthew Might and Olin Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, 18(5–6): 821–864, 2008.

Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 1–11. ACM, 2002.

David A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205 – 224, 2007.

Jens Nicolay, Coen De Roover, Wolfgang De Meuter, and Viviane Jonckers. Automatic parallelization of side-effecting higher-order scheme programs. In *Proceedings of the Eleventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2011)*, pages 185–194, Williamsburg, VA, USA, September 2011.

Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Determining dynamic coupling in JavaScript using object type inference. In *Proceedings of the Thirteenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013)*, pages 126–135, Eindhoven, NL, September 2013.

Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Detecting function purity in javascript. In *Proceedings of the Fifteenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)*, pages 101–110, Bremen, DE, September 2015.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Type and effect systems. In *Principles of Program Analysis*, pages 281–361. Springer, 1999.

Lance Norskog. Grid: A simple N-dimensional array package, 1993. URL ftp://ftp.cs.indiana.edu/pub/scheme-repository/code/struct/grid.scm.

Sungwoo Park and Robert Harper. A logical view of effects. *Unpublished manuscript*, 2004.

David J Pearce. Jpure: a modular purity system for java. In *Compiler Construction*, pages 104–123. Springer, 2011.

Mihalis Pitidis and Konstantinos Sagonas. Purity in erlang. In *Implementation and Application of Functional Languages*, pages 137–152. Springer, 2011.

Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, page 4. ACM, 2013.

Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for java programs. *Lecture notes in computer science*, pages 199–215, 2005.

D. Schmidt. Abstract interpretation and static analysis. http://santos.cis.ksu.edu/schmidt/Escuela03/WSSA/talk1p.pdf, 2003.

Michael I. Schwartzbach. Lecture notes on static analysis. 2008. URL http://www.brics.dk/~{}mis/static.html.

Jun Shirako, Hironori Kasahara, and Vivek Sarkar. Language extensions in support of compiler parallelization. In *Languages and Compilers for Parallel Computing*, pages 78–94. Springer, 2007.

Olin Shivers. Control flow analysis in scheme. In *ACM SIGPLAN Notices*, volume 23, pages 164–174. ACM, 1988.

Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990.

Guy L Steele Jr. *Rabbit: A compiler for Scheme*, volume 474. MIT Artificial Intelligence Laboratory Technical Report, 1978.

Quentin Stievenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Detecting concurrency bugs in higher-order programs through abstract interpretation. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 232–243. ACM, 2015.

Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1-2):11–49, 2000.

Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of functional programming*, 2(3):245–271, 1992.

Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *Theoretical Aspects of Computer Software*, pages 224–243. Springer, 1994.

Noah Van Es, Jens Nicolay, Quentin Stievenart, Theo D'Hondt, and Coen De Roover. A performant scheme interpreter in asm.js. In *To appear in Proceedings of the 31st ACM/SIGAPP Symposium on Applied Computing (SAC 2016)*. ACM, 2016.

David Van Horn and Matthew Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

Maarten Vandercammen, Jens Nicolay, Stefan Marr, Joeri De Koster, Theo D'Hondt, and Coen De Roover. A formal foundation for trace-based jit compilers. In *Proceedings of the 13th International Workshop on Dynamic Analysis*, pages 25–30. ACM, 2015.

Dimitrios Vardoulakis and Olin Shivers. Cfa2: A context-free approach to control-flow analysis. In *Programming Languages and Systems*, pages 570–589. Springer, 2010.

Jerome Vouillon and Pierre Jouvelot. Type and effect systems via abstract interpretation, 1995. URL `http://www.cri.ensmp.fr/classement/doc/A-273.pdf`.

Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, 2003.

John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. *ACM Sigplan Notices*, 34(10):187–206, 1999.