Automatically Proving the Correctness of Program Analyses and Transformations

Sorin Lerner

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2006

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Sorin Lerner

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

_____

Craig Chambers

Reading Committee:

_____

Craig Chambers

_____

Daniel Grossman

_____

Jan Vitek

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Automatically Proving the Correctness of Program Analyses and Transformations

Sorin Lerner

Chair of the Supervisory Committee:
Professor Craig Chambers
Computer Science and Engineering

In this dissertation, I describe a technique for automatically proving compiler optimizations sound, meaning that their transformations are always semantics-preserving. I first present a domain-specific language, called Rhodium, for implementing optimizations using local propagation and transformation rules that manipulate explicit dataflow facts. Then I describe a technique for automatically proving the soundness of Rhodium optimizations. The technique requires an automatic theorem prover to discharge a simple proof obligation for each propagation and transformation rule.

I have written a variety of forward and backward intraprocedural dataflow optimizations in Rhodium, including constant propagation and folding, branch folding, full and partial redundancy elimination, full and partial dead assignment elimination, an intraprocedural version of Andersen's points-to analysis, arithmetic-invariant detection, loop-induction-variable strength reduction, and redundant array load elimination. I implemented Rhodium's soundness-checking strategy using the Simplify theorem prover, and I have used this implementation to automatically prove that the Rhodium optimizations I wrote were sound. I implemented a prototype execution engine for Rhodium so that Rhodium optimizations can be directly executed. I also developed a way of interpreting Rhodium optimizations in both flow-sensitive and -insensitive ways, and of applying them interprocedurally given a separate context-sensitivity strategy, all while retaining soundness.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I want to thank my adviser Craig Chambers, for guiding and supporting me throughout my graduate career. Whenever I had a problem, whether technical or not, Craig was there to help me solve it. Just knowing that I could go to him for help, about any matter whatsoever, was a great source of reassurance for me. Craig has also been a fantastic research mentor. He has helped me better understand how to determine what's important and what's not, how to motivate research ideas, and how to write papers. Finally, Craig's ability to solve technical problems, to write papers, to hack code, to teach classes, and to advise students, all at the same time, was a great source of inspiration for me. I hope that one day I will become as prolific and inspiring an adviser as he is.

There are many other mentors who have shaped my career over the past six years. A huge thanks goes to Manuvir Das, who has constantly supported me, and has always been willing to make time to talk to me. My discussions with Norman Ramsey at conferences and when he visited at the University of Washington were very helpful in shaping my dissertation. Dan Grossman has also been a great source of advice, on both technical and non-technical matters. In addition, Dan has given me great comments and suggestions on the body of this dissertation, for which I am very grateful. Finally, I also want to thank Jan Vitek, who has recently been very helpful with my slow transition to becoming a faculty.

The work presented in this dissertation is the result of a long and fruitful collaboration with Todd Millstein and Craig Chambers. Todd was a great research partner throughout this project, and I learned a great deal from my interactions with him. I am very grateful to have worked with him, and I hope our paths will meet again in the future. A special thanks goes to Erika Rice for implementing arrays in the Rhodium intermediate language, and for the fantastic job she has done on the rule inferencer, a follow-up project to my dissertation.

Throughout my graduate career, I have also been involved in a variety of research

projects that were not directly related to my dissertation, and I want to thank all the collaborators I interacted with on these projects. In particular, thanks to David Grove for his contributions to the dataflow analysis framework, and a huge thanks to Manuvir Das and Mark Seigle for all the fun and exciting research we did together on the ESP project.

My graduate school experience would not have been the same were it not for all the enriching interactions with other graduate students, many of whom work in areas different from mine. I especially want to thank Krishna Gummadi, Stefan Saroiu, Keunwoo Lee, Todd Millstein, Matthai Phillipose, Jayant Madhavan, and Amlan Mukherjee for all the technical discussions we've had. They have allowed me to better understand other areas of sciences and engineering, and they also forced me to look at my own research from a different perspective. I also want to thank my office mates, Amol Prakash, Andrew Petersen, and Colin Zheng, for all the fun and interesting discussions we've had, going back to the days of Sieg 433. Even in the worst of times, their presence made going to the office lots of fun.

I also want to thank the Saturday night crowd, Amlan Mukherjee, Keunwoo Lee, Krishna Gummadi, and Sandra Fan, for the interesting and thought-provoking discussions we've had on any and every possible topic. A special thanks goes to Amlan Mukherjee and Sandra Fan for supporting me through some of the toughest parts of graduate school.

Finally, last but not least, I want to thank my parents. I often feel that I must have the best parents in the whole world. Their constant unwavering love and support have helped me tremendously throughout this PhD, far more than they realize, and far more than can be put in words.

Chapter 1

INTRODUCTION

1.1 Correctness of Program Analysis and Transformation Tools

The reliability of any software program depends on the reliability of all the tools that process it, including compilers, static and dynamic checkers, and source-to-source translators: a correct program compiled incorrectly is effectively incorrect; a guarantee of memory safety (no buffer overruns) or security safety (no high-security information flows to low-security variables), if provided by a buggy checker, is in fact no guarantee at all. Unfortunately, program analysis and transformation tools (PATs) can be difficult to develop, even for experts in the field of program analysis. It may take years before a new compiler is stable enough for programmers to trust it, and even then the compiler will probably still have numerous bugs. Aside from having an impact on software reliability in general, the difficulty of writing correct PATs also hinders the development of new languages and new architectures, and it discourages end programmers from extending PATs with domain-specific checkers or optimizers.

One of the most error-prone parts of a PAT are the analyses and transformations themselves. Each new program processed by a PAT potentially triggers new patterns of interactions among its analyses and transformations, and it is difficult to cover all these interactions in test suites. Furthermore, it is becoming less and less feasible to increase the reliability of a PAT by simply disabling most of its analyses and transformations. With the widespread adoption of systems whose good performance depends heavily on compiler optimizations – for example just-in-time compilers and higher-level languages like C# [50] and Java [8] – turning off the optimizer is no longer a reasonable option. With more and more systems aiming to provide strong guarantees, it is no longer possible to disable the static checkers that provide these very guarantees.

The research presented in this dissertation is aimed at making it easier to build reliable program analyses and transformations by providing the right abstractions for implementing them, while at the same time providing strong theoretical guarantees about their correctness. This dissertation focuses on the most commonly used PAT, namely the compiler, and the analyses and transformations found in a compiler, namely code optimizations. However, the techniques presented in this dissertation are applicable to program analyses and transformations in general, regardless of the PAT they are used in. Furthermore, this dissertation focuses on a particular aspect of correctness, namely soundness. In general, a soundness guarantee says that nothing bad will happen, whereas a correctness guarantee says that the implementation does what it should do. In the context of compilers, guaranteeing correctness would require having some sort of high-level specification stating what optimizations are meant to do, and then checking the implementation against these specifications. My work focuses on the simpler task of checking that compiler optimizations are sound, meaning that their transformations are always semantics-preserving.

1.2 Previous Approaches

The previously known techniques for improving the reliability of compilers can be categorized into three broad areas: testing, translation validation, and human-assisted soundness proofs. This section discusses the advantages and disadvantages of these three techniques, setting the stage for the next section, which describes the solution presented in this dissertation.

The simplest and most commonly used technique for gaining confidence in the correctness of a compiler is testing. Testing consists of running the compiler on various input programs and checking that the optimized version of each program produces correct results on various inputs. This method is simple and easy to understand, and it can be applied to any optimization. Testing can find many bugs, thus increasing one's confidence in the correctness of the compiler. However, testing cannot provide any guarantees: it does not guarantee the absence of bugs in the compiler, nor does it even guarantee that any particular optimized program is compiled correctly. It can also be tedious to assemble an extensive test suite of programs and program inputs with resulting program outputs.

Translation validation [91, 77, 126, 125, 45] improves on the testing approach by automatically checking, during compilation, whether or not the optimized version of an input program is semantically equivalent to the original program. Translation validation can therefore guarantee the soundness of certain compiler runs, but the compiler itself is still not guaranteed to be bug-free: there may still exist programs for which the compiler produces incorrect output, and there is little recourse for a programmer if a compilation cannot be validated. Furthermore, to make translation validation effective, one usually has to narrow the scope of optimizations being considered, thus reducing the generality of the approach. For example, until recently, only structure-preserving transformations could be validated effectively [77, 45]. Finally, since the validation is done during compilation, translation validation can also have a substantial impact on the time to run the compiler.

The best solution would be to prove the compiler sound, meaning that for any input program, the compiler always produces an equivalent output program. Optimizations, and sometimes even complete compilers, have been proven sound by hand [29, 30, 70, 68, 48, 83, 31, 61, 60, 108, 61, 15], or using human-assisted interactive theorem provers [124, 21, 1]. However, manually proving large parts of a compiler sound requires a lot of effort and theoretical skill on the part of the compiler writer. In addition, these proofs are usually done for abstract descriptions of optimizations, and bugs may still arise when the algorithms are implemented from the specification.

These three broad research areas of compiler correctness can therefore be summarized as follows: the testing approach is fully automated once test inputs have been generated, and it can be applied to any optimization, but it provides no soundness guarantees; translation validation is also fully automated, and in addition, it provides a per-compilation guarantee of soundness, but it is effective only on certain kinds of optimizations; and finally, human-assisted proofs are not automated, requiring substantial human effort, but they are very general, and they provide the best possible guarantee, a once-and-for-all guarantee of soundness. Table 1.1 summarizes the characteristics of these three broad research areas, and Figure 1.1 shows a pictorial representation of two dimensions of Table 1.1.

Table 1.1: The three previous approaches to compiler correctness

	Automation	Soundness guarantee	Generality
Testing	running tests can be fully automated, but generating test inputs and outputs is hard to automate	none	applicable to any optimization
Translation validation	fully automated	per-compilation	is effective only on certain kinds of optimizations
Human-assisted proofs	semi-automated, requiring a substantial amount of human effort	once and for all	applicable to any optimization, but the more complicated the optimization is, the more human effort is required
Goal of this dissertation	fully automated	once and for all	as general as possible, given the automation and soundness guarantee requirements

1.3 Statement of the Thesis

My thesis is that *it is possible to provide once-and-for-all guarantees of soundness in a fully automated way for a variety of program analyses and transformations, thus combining the benefits of translation validation and human-assisted soundness proofs.* Figure 1.1 shows pictorially where my thesis lies in the space of previous work. Unfortunately, reasoning automatically about the soundness of program analyses and transformations is hard. The proofs require induction, which is hard to automate in general, and the theorems to be proven contain alternating quantifiers, a pattern that is difficult to handle automatically. The difficulties in overcoming these problems have caused the majority of previous work on compiler soundness to either give up on full automation, resulting in human-assisted soundness proofs, or give up on once-and-for-all guarantees, resulting in translation vali-

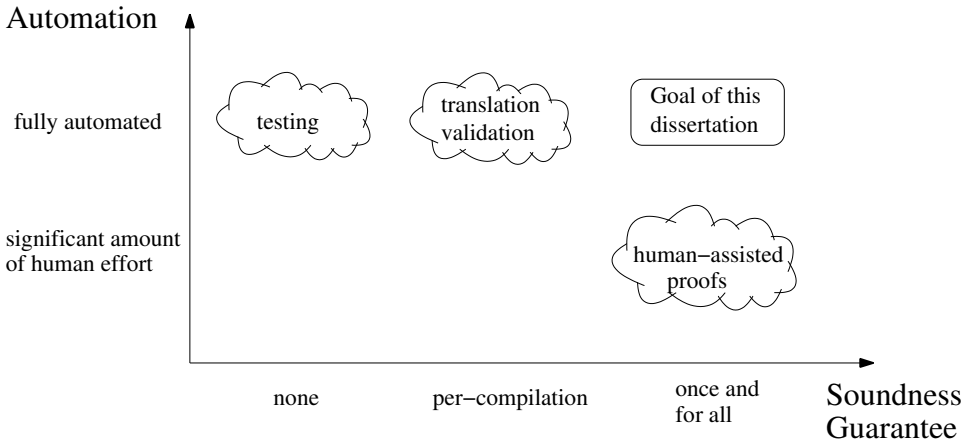


Figure 1.1: Pictorial representation of two dimensions of the space of previous work

dation. This dissertation shows that once-and-for-all guarantees of soundness can in fact be attained automatically. The key enabling idea is to use a *domain-specific language* for implementing compiler optimizations. The stylized form of this domain-specific language then makes it feasible to automatically check that optimizations are sound, meaning that their transformations are always semantics-preserving.

I therefore support my thesis in this dissertation as follows:

- I present a domain specific language called Rhodium for implementing program analyses and transformations over C-like programs [64, 65]. Aside from reducing the potential for errors by making program analyses and transformations easier to write and maintain, the restricted domain of Rhodium makes it amenable to rigorous static checking that would be impossible otherwise. In particular, I have implemented a tool called a soundness checker that leverages the stylized form of Rhodium analyses and transformations to check them for soundness automatically, before they are even run once. Once checked for soundness, Rhodium analyses and transformations are executed in the compiler by an execution engine that is trusted to be correct.

- I present a variety of program analyses and transformations that I implemented in Rhodium, and that I checked for soundness automatically using my soundness checker.

These include constant propagation and folding, copy propagation, common subexpression elimination, branch folding, partial redundancy elimination, partial dead assignment elimination, loop-invariant code motion, loop-induction-variable strength reduction, a flow-sensitive version of Andersen's points-to analysis [6] with heap summaries, arithmetic invariant detection, constant propagation through array elements, redundant array load elimination, and integer range analysis.

1.4 Contributions

The main contribution of this dissertation is the design of a language for writing program analyses and transformations that can be checked for soundness automatically. The challenge in designing such a language stems from the tension between expressiveness and checkability. The more expressive the language is, the more difficult it is to automatically reason about the soundness of analyses and transformations written in that language. I address this challenge in the Rhodium language by incorporating a variety of novel features that increase expressiveness, while still retaining automated soundness checking. The most important of these features are as follows:

- **Locally checked rules.** In Rhodium, programmers declare *dataflow facts* that represent useful information about a program, and then they write *local rules* for propagating these facts across statements and for using these facts to trigger program transformations. These rules are akin to regular dataflow functions with which compiler writers are already familiar [3, 74, 7, 82], and as a result they provide a natural and easy way of expressing complex optimizations. However, despite their expressiveness, the stylized form of these rules makes them amenable to fully automated soundness checking. To check an analysis or transformation for soundness, the programmer must provide a semantic meaning for each dataflow fact, in the form of a predicate over program states. The Rhodium system then asks the Simplify [36] automatic theorem prover to discharge a *local soundness lemma* for each rule, using the meanings of the facts manipulated by the rules and the concrete semantics of the program's statements. I proved, once by hand, that for any Rhodium analysis or trans-

formation, if the local soundness lemmas hold, then the analysis or transformation is globally sound. This split of the proof task between the human and the automatic theorem prover is critical. The manual proof takes care of the necessary induction over program execution traces, and it takes care of alternating quantifiers, both of which would be difficult to automate. As a result, the automatic theorem prover only needs to reason about noninductive lemmas that mention only one statement at a time.

- **Soundness vs. profitability.** In many compiler optimizations, the condition that specifies when a transformation is *legal* can be separated from the condition that specifies when a transformation is *profitable*. Rhodium provides *profitability facts* for implementing arbitrarily complex profitability decisions without affecting the soundness of an optimization. As a result, the profitability part of an optimization does not need to be reasoned about and can be written in a general-purpose language, thereby removing any limitations on its expressiveness. This way of factoring out the profitability part of an optimization from the rest of the optimization is critical for automatically proving the soundness of complex optimizations. Without profitability facts, the extra complexity added to the Rhodium rules to express profitability information would prevent automated soundness reasoning.

- **Dynamic semantics extensions.** Rhodium allows the programmer to define "virtual" extensions to the dynamic semantics of the intermediate language. These extensions can compute properties of program execution traces, for example, the statement at which each memory location was allocated. These extensions can then be referenced in the meanings of dataflow facts, for instance in a points-to analysis with allocation-site heap summaries. The net effect of dynamic semantics extensions is that they allow meanings that would otherwise mention full traces to mention only the current program state. As a result, they make the theorem proving task simpler, and thereby allow for a wider class of optimizations to be proven sound automatically.

1.5 Outline

The next chapter of this dissertation, Chapter 2, provides an overview of the Rhodium system, and is structured exactly along the lines of the above contributions. In particular, Section 2.1 presents the Rhodium language using a few examples, and then the following three sections explain the above contributions in more detail: Section 2.2 presents an overview of the Rhodium proof strategy, Section 2.3 shows how profitability facts can be used to separate soundness from profitability, and finally Section 2.4 presents dynamic semantics extensions. The chapter concludes with a section discussing termination of Rhodium optimizations, an important but less fundamental aspect of the Rhodium system. Chapter 2 keeps formal details to a minimum and tries to provide intuitive explanations.

The succeeding four chapters describe in greater depth the ideas already presented in Chapter 2. These chapters, which are more formal than Chapter 2, constitute the main technical content of this dissertation. In particular, Chapter 3 describes Rhodium IL programs, which are the programs that Rhodium optimizations manipulate. Chapter 4 presents a common theoretical framework for defining and proving the soundness of program analyses and transformations over Rhodium IL programs. Chapters 5 and 6 then use this framework to formalize the semantics of forward and backward Rhodium optimizations, and to show how Rhodium optimizations are checked for soundness automatically.

Chapter 7 then describes how to execute Rhodium optimizations. It describes an intraprocedural flow-sensitive execution engine for Rhodium optimizations, and it also shows how Rhodium analyses can be run in a flow-insensitive and/or interprocedural mode.

Chapter 8 evaluates the Rhodium system along several dimensions. Chapter 9 describes related work. Chapter 10 presents several directions for future work, and concludes.

Chapter 2

OVERVIEW OF RHODIUM

2.1 Rhodium by example

2.1.1 Constant propagation in Rhodium

Rhodium optimizations run over a C-like intermediate language (IL) with functions, recursion, pointers to dynamically allocated memory and to local variables, and arrays. The optimizations manipulate a control-flow graph (CFG) representation of the IL program, with each node representing a simple register-transfer-level statement.

The purpose of an optimization is to compute information about an IL program and then to use this information to perform optimizing transformations. One can therefore identify three main steps in writing an optimization: (1) defining how to represent the desired information; (2) defining how to compute this information; and (3) defining how to use this information to perform transformations. This section illustrates these three steps in Rhodium using a simple constant propagation example, shown in Figure 2.1.[1] The purpose of this optimization is to determine which variables contain compile-time constants and to replace all uses of these variables with their constant values.

Step 1: Dataflow fact declarations

Information about an IL program is encoded in Rhodium by means of dataflow facts, which are user-defined function symbols (in the logic sense) applied to a set of terms, for example $hasConstValue(\texttt{x}, \texttt{5})$ or $exprIsAvailable(\texttt{x}, \texttt{a} + \texttt{b})$. During execution of a Rhodium optimization, each edge in the CFG will be annotated with a set of these facts. The Rhodium programmer defines what kind of facts are going to be computed by means of *fact schemas*. A fact schema is a parametrized dataflow fact (a pattern, in essence) that can be instantiated to create actual dataflow facts. For example, the constant propagation optimization

[1] For the complete syntax of the Rhodium language, see Figures 5.1, 5.2, 5.3 and 3.1.

```
1.  define forward edge fact hasConstValue(X:Var, C:Const)
2.  with meaning η(X) = C
3.  decl X:Var, Y:Var, C:Const
4.  if currStmt = [X := C]
5.  then hasConstValue(X, C)@out
6.  if currStmt = [Y := E] ∧
7.     hasConstValue(X, C)@in ∧
8.     X ≠ Y
9.  then hasConstValue(X, C)@out
10. if currStmt = [Y := X] ∧
11.    hasConstValue(X, C)@in
12. then transform to Y := C
```

Figure 2.1: A simple constant propagation optimization in Rhodium

of Figure 2.1 contains a fact-schema declaration on line 1. This declaration states that the programmer wants to compute facts of the form $hasConstValue(X, C)$, where X ranges over variables in the IL program being optimized, and C ranges over constants. Intuitively, the presence of a $hasConstValue(X, C)$ fact on an edge in the CFG is meant to represent the fact that the variable X has the constant value C. This intuition is made precise with the meaning declaration on line 2. The meaning of a fact schema defines what the programmer intended instances of that fact schema to capture about the run-time behavior of the IL program. The details of this meaning can be ignored for now, as they will be covered in Section 2.2. Meanings are not used to execute the Rhodium optimizations; they are only used to check them for soundness.

Step 2: Propagation rules

Programmers define how to compute dataflow facts in Rhodium using *propagation rules*, which are a stylized way of writing traditional flow functions. Propagation rules in Rhodium simply indicate how facts are propagated across CFG nodes. For example, the rule on lines 6–9 of Figure 2.1 defines a condition for preserving a *hasConstValue* fact across sim-

ple assignments: if a fact $hasConstValue(X, C)$ appears on the incoming CFG edge of an assignment $Y := E$, and the assignment does not modify X (that is, $X \neq Y$), then the dataflow fact $hasConstValue(X, C)$ should appear on the outgoing edge of the assignment.

The part of a rule immediately after the "**if**" is called the *antecedent* and the part after the "**then**" is the *consequent*. Each propagation rule is interpreted within the context of a CFG node. The special variable *currStmt* refers to the IL statement at the current CFG node. Dataflow facts are followed by @ signs, with the name after the @ sign indicating the edge on which the fact appears. For example, $hasConstValue(X, C)@in$ is true if the incoming CFG edge of the current node is annotated with a fact of the form $hasConstValue(X, C)$.

The semantics of a propagation rule on a CFG is as follows: for each substitution of the rule's free variables that make the antecedent valid at some node in the CFG, the fact in the consequent is propagated. For the rule described above, a fact of the form $hasConstValue(X, C)$ will be propagated on the outgoing edge of a node for each substitution of X and C with variables and constants that makes the antecedent valid.

While the rule on lines 6–9 of Figure 2.1 specifies how to preserve *hasConstValue* facts, the rule on lines 4–5 specifies how to introduce them in the first place. That rule says that the outgoing CFG edge of a statement of the form $X := C$ should be annotated with a fact of the form $hasConstValue(X, C)$.

A set of propagation rules together implicitly define a dataflow analysis $\mathcal{A}$ whose domain D is the powerset lattice of all dataflow facts: $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot) = (2^{Facts}, \cap, \cup, \supseteq, \emptyset, Facts)$, where *Facts* is the set of all fact-schema instances. Each edge in the CFG is therefore annotated with a set of dataflow facts, where bigger sets are more precise than smaller sets.[2] The analysis starts with all edges in the CFG set to $\bot$, except for the input edge, which is initialized to $\top$. The flow function F of the analysis is defined by the propagation rules: given a node and a set of incoming dataflow facts, F returns the set of all dataflow facts propagated by any of the individual rules.

The solution of the induced analysis $\mathcal{A}$ is the least fixed point of the standard set of dataflow equations generated from F. Although the two rules in Figure 2.1 propagate

[2]I use the abstract interpretation convention that $\bot$ is the most optimistic information, and $\top$ is the most conservative information.

facts that are instances of the same fact schema, different rules can propagate instances of different fact schemas, and the fixed point is computed over all fact schemas simultaneously.

Step 3: Transformation rules

Rhodium propagation rules are used to define dataflow analyses. The information computed by these analyses can then be used in *transformation rules* to optimize IL programs. A transformation rule describes the conditions under which a node in the CFG can be replaced by a new node without changing the behavior of the program. As an example, the transformation rule on lines 10–12 of Figure 2.1 states that a statement $Y := X$ should be transformed to $Y := C$ if it is known that X has the constant value C.

Together, the propagation and transformation rules of Figure 2.1 define a simple constant propagation optimization. The propagation rules compute which variables have constants, and the transformation rule uses this information to simplify variables to constants.

2.1.2 Node facts

Each one of the propagation rules from Figure 2.1 applied to only one statement kind at a time. Unfortunately, this coding style leads to many repetitive rules. Consider, for example, the propagation rule from lines 6–9 of Figure 2.1:

if $currStmt = [Y := E] \wedge$
  $hasConstValue(X, C)@in \wedge$
  $X \neq Y$
then $hasConstValue(X, C)@out$

One would have to write a similar rule for `skip` statements:

if $currStmt = [\texttt{skip}] \wedge$
  $hasConstValue(X, C)@in \wedge$
then $hasConstValue(X, C)@out$

And one for branch nodes in the CFG (which are represented by `if` statements):

if $currStmt = [\texttt{if } A \texttt{ goto } L_1 \texttt{ else } L_2] \wedge$
  $hasConstValue(X, C)@in \wedge$
then $hasConstValue(X, C)@out$

In fact, one would have to write a similar rule for every statement form that does not modify X. Ideally, however, the programmer would have to write only one such preservation rule, which abstracts over the various statement kinds. This rule would simply state that $hasConstValue(X, C)$ is preserved through any statement that does not modify X. Unfortunately, the Rhodium facts presented so far have all been *edge facts*, capturing information about edges in the CFG, not statements. In order to facilitate writing rules that apply to multiple statement kinds, Rhodium provides *node facts* to capture properties of statements. An an example, the programmer can define a node-fact schema $mustNotDef(X)$, which captures the fact that a node does not modify X. The preservation rule for *hasConstValue* would then become:

if $hasConstValue(X, C)@in \wedge mustNotDef(X)$
then $hasConstValue(X, C)@out$

Node facts can easily be distinguished from edge facts in a rule because node facts do not have @ signs following them.

Programmers define node-fact schemas in Rhodium by providing a predicate over a statement, referred to in the predicate's body using the distinguished variable *currStmt*. Figure 2.2 shows the definition of two node-fact schemas: *stmt* and *mustNotDef*. The $stmt(S)$ fact schema simply says that the statement at the current node is S. The $mustNotDef(Y)$ fact schema is slightly more complicated, as it performs a case analysis on the current statement: $mustNotDef(Y)$ holds at a node if the current statement is a declaration of or an assignment to a variable different from Y (the statement `decl` $X[I]$ declares an array X of size I); $mustNotDef(Y)$ does not hold for pointer stores (since the intermediate language allows taking the address of a local variable) or procedure calls (since the procedure may be passed pointers from which the address of Y is reachable); finally, it holds for all other statement forms (namely, conditionals and skip – see Figure 3.1 for the complete syntax of the Rhodium IL). The "**case**" predicate is a convenience that provides a form of pattern matching, but it is easily desugared into an ordinary logical expression. Figure 2.3 shows the Rhodium code for constant propagation, this time taking advantage of node facts. Not

```
1.  define node fact stmt(S:Stmt) ≜ currStmt = S
2.  define node fact mustNotDef(Y:Var) ≜
3.    case currStmt of
4.         decl X      ⇛  X ≠ Y
5.         decl X[I]   ⇛  X ≠ Y
6.         X := E      ⇛  X ≠ Y
7.         *X := Z     ⇛  false
8.         X := P(Z)   ⇛  false
9.         else        ⇛  true
10.   endcase
```

Figure 2.2: Node facts in Rhodium

only is the code that uses node facts easier to understand, but it also defines a version of constant propagation that is more complete than the one from Figure 2.3, with little additional programming effort.

2.1.3 Standalone analyses

The propagation rules that we have seen so far were used to compute facts for the purpose of triggering transformations. Analyses can also be written to compute facts that will be useful in writing other propagation rules. An example of such a standalone analysis is pointer analysis, which determines what variables point to what other variables. This information is often useful in defining propagation rules. For example, on line 7 of Figure 2.2, the definition of *mustNotDef* could be made less conservative if only we knew that X did not point to Y.

Figure 2.4 shows a pointer analysis in Rhodium that computes exactly this kind of information: it determines when a variable does not point to another. Because Rhodium's strategy for automated soundness checking is geared toward *must* analyses, the pointer information in Figure 2.4 is encoded using the must-not-point-to relation instead of the more traditional may-point-to relation. Programmers can recover the may-point-to relation using *virtual* dataflow facts, which are used in Rhodium to define shorthands for boolean combinations of other facts. This facility allows a $mayPointTo(X, Y)$ fact to be defined as

1. **define forward edge fact** $hasConstValue(X{:}Var, C{:}Const)$
2. **with meaning** $\eta(X) = C$
3. **decl** $X{:}Var, Y{:}Var, C{:}Const$
4. **if** $stmt(X := C)$
5. **then** $hasConstValue(X, C)@out$
6. **if** $hasConstValue(X, C)@in \wedge mustNotDef(X)$
7. **then** $hasConstValue(X, C)@out$
8. **if** $stmt(X := Y) \wedge hasConstValue(Y, C)@in$
9. **then transform to** $X := C$

Figure 2.3: Constant propagation using node facts

$\neg mustNotPointTo(X, Y)$, as shown on lines 5–6 of Figure 2.4.

Negation is provided in Rhodium only as a convenience. After all the virtual facts have been expanded out, and negation has been pushed to the inside (using DeMorgan's law) through conjunctions, disjunctions and quantifiers, all negation on edge facts are required to cancel out. The absence of negated edge facts guarantees the monotonicity of the implicitly defined flow function F, as further discussed in Chapter 5. Although disallowing negated edge facts sounds restrictive, it actually corresponds to a common usage pattern. Because Rhodium facts are all *must* facts, the absence of a fact does not provide any information – only its presence does. As a result, I never found the need to use any negated edge facts, except as a notational convenience. For example, in analyses that use $mayPointTo(X, Y)$, it is always the *lack* of possible points-to information, i.e., $\neg mayPointTo(X, Y)$, that enables more-precise analysis or transformation, which when expanded yields $mustNotPointTo(X, Y)$.

The rules in Figure 2.4 range from very simple to fairly complex. The first two rules are the analogues of the introduction rule and the preservation rule from constant propagation. In particular, the rule on lines 8–9 specifies how to introduce *mustNotPointTo* facts: it says that the outgoing CFG edge of a statement $X := \&Z$ should be annotated with all facts of the form $mustNotPointTo(X, Y)$, where Y and Z are distinct variables.

1. **define forward edge fact** $mustNotPointTo(X{:}Var,\ Y{:}Var)$
2. **with meaning** $\eta(X) \neq \eta(\&Y)$
3. **define forward edge fact** $mustPointToSomeVar(X{:}Var)$
4. **with meaning** $\exists Z : Var\ .\ \eta(X) = \eta(\&Z)$
5. **define virtual edge fact**
6. $\quad mayPointTo(X{:}Var,\ Y{:}Var) \triangleq \neg mustNotPointTo(X, Y)$
7. **decl** $X{:}Var,\ Y{:}Var,\ Z{:}Var,\ A{:}Var,\ B{:}Var$
8. **if** $stmt(X := \&Z) \wedge Y \neq Z$
9. **then** $mustNotPointTo(X, Y)@out$
10. **if** $mustNotPointTo(X, Y)@in \wedge mustNotDef(X)$
11. **then** $mustNotPointTo(X, Y)@out$
12. **if** $stmt(X := A)\ \wedge mustNotPointTo(A, Y)@in$
13. **then** $mustNotPointTo(X, Y)@out$
14. **if** $stmt(*A := B) \wedge$
15. $\quad mustPointTo(A, X)@in \wedge$
16. $\quad mustNotPointTo(B, Y)@in$
17. **then** $mustNotPointTo(X, Y)@out$
18. **if** $stmt(*A := B) \wedge$
19. $\quad mustNotPointTo(X, Y)@in \wedge$
20. $\quad mustNotPointTo(B, Y)@in$
21. **then** $mustNotPointTo(X, Y)@out$
22. **if** $stmt(X := *A) \wedge$
23. $\quad mustPointToSomeVar(A)@in \wedge$
24. $\quad \forall\ B{:}Var\ .\ mayPointTo(A, B)@in \Rightarrow$
25. $\qquad\qquad mustNotPointTo(B, Y)@in$
26. **then** $mustNotPointTo(X, Y)@out$

Figure 2.4: Pointer analysis in Rhodium

And the rule on lines 10–11 specifies how to preserve *mustNotPointTo* facts: if the fact $mustNotPointTo(X, Y)$ appears on the incoming CFG edge of a node n, and n does not modify X, then the dataflow fact $mustNotPointTo(X, Y)$ should appear on the outgoing edge of n.

The rule on lines 12–13 shows how to propagate pointer information through simple assignments. The outgoing information, $mustNotPointTo(X, Y)$, is a different instantiation of the *mustNotPointTo* fact schema than the incoming information, $mustNotPointTo(A, Y)$. This way of stringing together different dataflow facts allows the programmer to express complicated global conditions over the entire CFG, using only simple local propagation rules.

The rule on lines 14–17 shows how to propagate information through pointer stores. The $mustPointTo(A, X)$ fact, computed by rules not shown here, says that A definitely points to X. This rule shows how instances of different fact schemas can be mixed and matched in the antecedent to create complicated conditions for triggering the propagation of a new fact.

The above rule for pointer stores performs a strong update, where it is known exactly what A points to. It is also possible to write a weak-update rule for pointer stores, as shown on lines 18–21. The intuition behind this rule is as follows: if A points to X, then the assignment is storing B in X, in which case $mustNotPointTo(B, Y)@in$ guarantees $mustNotPointTo(X, Y)@out$; on the other hand, if A does *not* point to X, then the assignment does not modify X, in which case $mustNotPointTo(X, Y)@in$ guarantees $mustNotPointTo(X, Y)@out$.

Finally, lines 22–26 show a rule for propagating pointer information through pointer loads. This rule illustrates some of the additional logical connectives available in the antecedent, for example universal quantifiers and implication. In general, antecedents are expressed in an executable subset of first-order logic, in which quantifiers range over domains that are known at execution time to be finite and small enough to iterate over. For example, a quantifier ranging over the infinite set of integers, or the finite-but-too-large set of 32-bit integers, would not be allowed, whereas a quantifier ranging over variables in the program being compiled would be allowed (see Section 2.5.1 for more details regarding

1. **define backward edge fact** $dead(X{:}Var)$
2. **with meaning** $\eta_1/X = \eta_2/X$
3. **decl** $X{:}Var$, $Y{:}Var$, $E{:}Expr$
4. **if** $stmt(X := E) \wedge mustNotUse(X)$
5. **then** $dead(X)@in$
6. **if** $stmt(\texttt{return}\ Y) \wedge X \neq Y$
7. **then** $dead(X)@in$
8. **if** $dead(X)@out \wedge mustNotUse(X)$
9. **then** $dead(X)@in$
10. **if** $stmt(X := E) \wedge dead(X)@out$
11. **then transform to** `skip`

Figure 2.5: Dead assignment elimination in Rhodium

termination of rules). The $mustPointToSomeVar(A)$ fact, whose rules are not shown here, says that A must point to some variable (and therefore does not point to the heap). The rule as a whole says that X does not point to Y after a statement $X := *A$ if all the variables in the may-point-to set of A do not point to Y.[3]

As a whole, the rules from Figure 2.4 show how a Rhodium analysis can easily be extended by simply writing new propagation rules. Starting with a basic pointer analysis, comprised of only the introduction and preservation rules, and extending it step by step with additional rules, we have now expressed in Rhodium a flow-sensitive intraprocedural version of Andersen's pointer analysis [6]. Furthermore, because each rule can be written and reasoned about independently, a novice programmer can easily extend this analysis, by adding rules to cover more cases, without having to understand the workings of existing rules.

2.1.4 Backward optimizations

All the rules presented so far have been *forward*: the antecedent refers only to a node's incoming CFG edge and the consequent refers only to a node's outgoing CFG edge. Rhodium also supports backward rules, where the antecedent refers only to the *out* edge and the consequent refers only to the *in* edge. Figure 2.5 shows an example of a backward optimization in Rhodium, namely dead assignment elimination. The goal of dead assignment elimination is to remove assignments to dead variables, which are variables that are not going to be used downstream of the assignment. The "deadness" of a variable X is encoded in Rhodium with a $dead(X)$ fact schema, shown on line 1. The meaning of $dead(X)$ on line 2 can be ignored for now – it will be explained in Section 6.2.3, along with the notation $\eta_1/X = \eta_2/X$. The first rule (on lines 4–5) says that a variable X is dead right before an assignment to X, as long as the right-hand side of assignment does not use X. The second rule (on lines 6–7) says that a variable X is dead right before a return statement, as long as the returned variable is different from X. The third rule (on lines 8–9) preserves the $dead(X)$ fact backwards through any statement that does not use X. Finally, the last rule (on lines 10–11) performs the actual transformation, removing any assignment to a dead variable. Statement removal is expressed in Rhodium by replacement with a `skip` statement. The execution engine for Rhodium optimizations does not actually insert such `skip`s.

Although Rhodium supports both forward and backward rules, it does not support bi-directional rules, in which the antecedent refers to both incoming and outgoing edge facts. Furthermore, Rhodium does not support bi-directional analyses, in which forward rules and backward rules interact: all forward rules are run together, and all backward rules are run together, but the forward rules and the backward rules cannot run simultaneously.

2.1.5 Indexed edge names

The edge names presented so far have been *in* and *out*. Some CFG nodes, however, have more than one incoming/outgoing edge. For example, a branch statement has two outgoing

[3]Note that, after expanding $a \Rightarrow b$ to $\neg a \vee b$, and $mayPointTo(X, Y)$ to $\neg mustNotPointTo(X, Y)$, the inner quantifier becomes $\forall B : Var\ .\ mustNotPointTo(A, B)@in \vee mustNotPointTo(B, Y)@in$, which does not contain any negated edge facts.

1. **define forward edge fact** *leq*(*X*:*Expr*, *Y*:*Expr*)
2. **with meaning** $\eta(X) \leq \eta(Y)$
3. **define forward edge fact** *gt*(*X*:*Expr*, *Y*:*Expr*)
4. **with meaning** $\eta(X) > \eta(Y)$
5. **decl** *X*:*Expr*, *Y*:*Expr*, L_1:*Label*, L_2:*Label*
6. **if** *stmt*(`if` *X* `<=` *Y* `then goto` L_1 `else` L_2)
7. **then** $leq(X, Y)@out[0]$
8. **if** *stmt*(`if` *X* `<=` *Y* `then goto` L_1 `else` L_2)
9. **then** $gt(X, Y)@out[1]$

Figure 2.6: Example of indexed edge names

edges, and a merge node has two incoming edges. In such cases, Rhodium programmers can use edge indices to indicate exactly what edge they are referring to: $in[i]$ refers to the i^{th} CFG input edge, and $out[i]$ refers to the i^{th} CFG output edge. For example, Figure 2.6 shows some Rhodium code that uses edge indices to propagate different facts along the two outgoing edges of a branch. The rule on lines 6–7 propagates $leq(X, Y)$ on the true branch of `if` *X* `<=` *Y* `then goto` L_1 `else` L_2, whereas the rule on lines 8–9 propagates $gt(X, Y)$ on the false branch.

Rhodium provides some syntactic sugar that can make the code from Figure 2.6 easier to read and write. First, $out[true]$ and $out[false]$ are interpreted as $out[0]$ and $out[1]$. Second, Rhodium supports rules of the form **if** ψ **then** $f_1(\ldots)@e_1 \wedge f_2(\ldots)@e_2 \wedge \ldots \wedge f_n(\ldots)@e_n$. Such a rule gets desugared into n rules, the i^{th} of which is **if** ψ **then** $f_i(\ldots)@e_i$. Using these syntactic sugars, the rules on lines 6–9 of Figure 2.6 can then be written as:

if *stmt*(`if` *X* `<=` *Y* `then goto` L_1 `else` L_2)
then $leq(X, Y)@out[true] \wedge gt(X, Y)@out[false]$

Indexed edge names can also be used for the *in* edge. The only statement in the Rhodium intermediate language that has more than one input edge is the `merge` statement, which has

two input edges. This statement is used at merge points in the CFG of the program being optimized. By default, the flow function for `merge` statements returns the intersection of the two incoming sets of dataflow facts, but programmers can override this default behavior by writing propagation rules for the `merge` statement, using $in[0]$ and $in[1]$ to refer to the two incoming edges. More details regarding user-defined merges, including examples, can be found in Section 2.5.3.

With the more general setting of indexed edge names in mind, there are several possible choices for defining what the unindexed *in* and *out* mean. The simplest possibility is to have *in* and *out* be syntactic sugar for $in[0]$ and $out[0]$. This approach is appealing because of its simplicity, but it interacts poorly with rules that abstract over many statement kinds. In particular, consider the following rule from constant propagation:

if $hasConstValue(X, C)@in \wedge mustNotDef(X)$
then $hasConstValue(X, C)@out$

If *out* were syntactic sugar for $out[0]$, then this rule would only propagate to the 0^{th} outgoing CFG edge, which, in the case of branch statements, would mean that $hasConstValue(X, C)$ is only propagated to the true side of the branch, not the false side. To propagate $hasConstValue(X, C)$ on the false side as well, one would have to write the additional rule:

if $stmt(\texttt{if } X \texttt{ <= } Y \texttt{ then goto } L_1 \texttt{ else } L_2) \wedge hasConstValue(X, C)@in$
then $hasConstValue(X, C)@out[false]$

This asymmetry arises from the fact that desugaring *out* to $out[0]$ is inherently asymmetrical, favoring the 0^{th} edge over the 1^{st} edge. The semantics of *in* and *out* used in the Rhodium system avoids this asymmetry. In particular, for forward rules, $f@out$ indicates that f is propagated on *each* outgoing edge, and $f@in$ means that f appears on *all* incoming edges; and for backward rules, $f@in$ indicates the f is propagated on each incoming edge, and $f@out$ indicates that f appears on all outgoing edges. This allows the programmer to write rules that abstract over statement kinds having a varying number of incoming/outgoing edges, without the asymmetry problems mentioned above. For example, the natural constant propagation rule, which propagates $hasConstValue(X, C)@out$, now does exactly the

right thing on branch statements. The ability to write such rules has been very useful in practice.

The semantics of *in* and *out* can still be given by desugaring them to indexed edges names, but the desugaring is more complicated. In particular, *in* and *out* are desugared by generating, for each propagation rule r, a specialized version of r for each input-output edge pair of each statement type. A formal description of this expansion is given in Appendix A.

Another issue that arises with indexed edge names is determining whether or not an edge index is in range. In order to facilitate this in-range check, I have chosen to disallow edge indices from being used in rules that apply to multiple statement kinds. If a rule applies to only one statement kind, then the number of incoming and outgoing edges is known, and as a result it becomes easy to check that indices are in range. Furthermore, edge indices are not allowed in transformation rules. Although these restrictions are more onerous than necessary, they have not been a source of problems in practice.

2.2 Proving soundness automatically

My goal is to ensure automatically that a Rhodium optimization is sound, according the following informal definition:

Def 1 *A Rhodium optimization O, which includes any number of propagation rules and transformation rules, is sound iff for all IL procedures P, the optimized version P' of P, produced by performing the transformations suggested by O, has the same semantics as P.*

The automatic proof strategy used in the Rhodium system separates the proof that O is sound into two parts: the first part is optimization *dependent* and it is discharged by an automatic theorem prover; the second part is optimization *independent* and it was shown by hand once and for all. For the optimization-dependent part, I define a sufficient soundness property that must be satisfied by each propagation or transformation rule in isolation, and I ask an automatic theorem prover to discharge this property for each rule. Separately, I have shown manually that if all propagation and transformation rules of an optimization satisfy the soundness property, then the optimization is sound. In the process of doing this, I have also shown that if all the propagation rules are sound, then the induced analysis is sound, meaning that its solution conservatively approximates the run-time behavior of the

program. This result can be useful by itself, for example if one wants to use facts computed by some propagation rules for a purpose other than optimizing transformations (say for reporting errors to users).

The formalization of Rhodium, including this manual proof, is based on a previous abstract-interpretation-based framework for composing dataflow analyses and transformations [63]. As a result, all Rhodium analyses and transformations can be composed, allowing them to interact in mutually beneficial ways.

The next two subsections present the local soundness condition for forward propagation rules and forward transformation rules, respectively. The description here is informal, and is meant to provide an intuition for how the soundness checker works. The formal details, along with a descriptions of the soundness conditions for backward rules, can be found in Chapters 4, 5 and 6.

2.2.1 Propagation rules

The definition of soundness of a propagation rule depends on *meaning* declarations that describe the concrete semantics of edge facts. The meaning of a fact f is a predicate on concrete execution states, η, with the intent that whenever f appears on an edge, the meaning of f should hold for all concrete execution states η the program could be in when control reaches that edge. For example, the meaning of $\mathit{hasConstValue}(X, C)$, shown on line 2 of Figure 2.1, is $\eta(X) = C$, where $\eta(E)$ represents the result of evaluating expression E in execution state η. The meaning of *hasConstValue* therefore says that the value of X in the execution state η should be equal to C. Meanings are provided at the fact-schema level, and they get instantiated in the same way as fact schemas. So, because the meaning of the fact schema $\mathit{hasConstValue}(X, C)$ is $\eta(X) = C$, the meaning of the fact $\mathit{hasConstValue}(\mathtt{x}, 3)$ is $\eta(\mathtt{x}) = 3$.

As another example, the meaning of $\mathit{mustNotPointTo}(X, Y)$, shown on line 2 of Figure 2.4, is $\eta(X) \neq \eta(\&Y)$, which says that the value of X in the execution state η should not be equal to the address of Y. Finally the $\mathit{mustPointToSomeVar}(X)$ declaration in Figure 2.4 shows an example of a more complicated meaning. The meaning of $\mathit{mustPointToSomeVar}(X)$, shown on line 4 of Figure 2.4, is $\exists Z : \mathit{Var} \;.\; \eta(X) = \eta(\&Z)$,

which says that there exists some variable that X must point to. In general, programmers can use the full power of first-order logic to express meanings.

To be sound, a propagation rule must preserve meanings: if a rule fires at a CFG node n, and the meanings of all facts flowing into n hold for execution states right before n, then the meaning of the propagated fact must hold for execution states right after n. This is stated informally in the following definition:

Def 2 *A propagation rule is said to be sound iff it satisfies the following property:*
For all concrete execution states η and CFG nodes n, if (1) the rule fires at node n, (2) the meanings of all facts flowing into n hold for η, and (3) the execution of n from η yields η', then the meaning of the propagated fact must hold for η'. (prop-sound)

For each propagation rule, the soundness checker uses the Simplify [36] automatic theorem prover to discharge (prop-sound). For example, consider the rule on lines 6–7 of Figure 2.3. The soundness checker effectively asks the theorem prover to show that if a statement satisfying $mustNotDef(X)$ is executed from a state η in which $\eta(X) = C$, then $\eta'(X) = C$ in the resulting state η'. The truth of this formula follows easily from the user-provided definition of *mustNotDef* and the system-provided concrete semantics of the Rhodium IL.

If all propagation rules are sound, then it can be shown by hand, once and for all, that the flow function F is sound. The definition of soundness of F is the one from a previous framework for composing dataflow analyses [63]. This definition depends on an *abstraction function* $\alpha : D_c \to D$, which formalizes the notion of approximation. The concrete semantics of the Rhodium IL is a collecting semantics, so that elements of D_c are sets of concrete stores. Meaning declarations naturally induce an abstraction function α: given a set $c \in D_c$ of concrete stores, $\alpha(c)$ returns the set of all dataflow facts whose meanings hold of all stores in c. An element $d \in D$ approximates an element $c \in D_c$ if $\alpha(c) \sqsubseteq d$, or equivalently if the meanings of all facts in d hold of all stores in c. The definition of soundness of F, adapted from [63], is then as follows (where F_c is the concrete collecting semantics flow function):

Def 3 *A flow function F is said to be sound iff it satisfies the following property:*

$$\forall\ (n, c, d) \in Node \times D_c \times D\ .\ \alpha(c) \sqsubseteq d \Rightarrow \alpha(F_c(n, c)) \sqsubseteq F(n, d)$$

The following lemma, which is formalized in Chapters 5 and 6, and proved in Appendices C and D, provides the link between the soundness of local propagation rules and the soundness of F.

Lemma 1 *If all propagation rules are sound, then the induced flow function F is sound.*

Once we know that the flow function F is sound, we can use the following definition and theorem from the framework on composing dataflow analyses to show that the analysis $\mathcal{A}$ is sound, meaning that its solution conservatively approximates the solution of the collecting semantics of the IL:

Def 4 *An analysis $\mathcal{A}$ is said be sound iff for any IL program P, and for any edge e in the CFG of P, the concrete solution c at edge e (computed by the collecting semantics) and the abstract solution d at edge e (computed by the analysis $\mathcal{A}$) are related by $\alpha(c) \sqsubseteq d$.*

Theorem 1 *If the flow function F is sound, then the analysis $\mathcal{A}$ induced by the standard dataflow equations of F is sound.*

A formalization of Definition 4 and Theorem 1 can be found in Chapter 4 and a proof of Theorem 1 can be found in Appendix B. The following theorem is immediate from Lemma 1 and Theorem 1:

Theorem 2 *If all propagation rules are sound, then the analysis $\mathcal{A}$ induced by the propagation rules is sound.*

Theorem 2 summarizes the part of the soundness proof of $\mathcal{A}$ that was done by hand once and for all. The automatic theorem prover is used only to discharge (prop-sound) for each propagation rule, thus establishing the premise of Theorem 2 that all propagation rules are sound. This way of factoring the proof is critical to automation. The proof of Theorem 2 (which includes proofs of Lemma 1 and Theorem 1) is relatively complex. It requires reasoning about F, α and a fixed-point computation, each one adding extra complexity. The proof also requires induction, which would be difficult to fully automate. In contrast, (prop-sound) is a non-inductive local property that requires reasoning only about a single state transition at a time. I have found that the heuristics used in automatic theorem provers are well-suited for these kinds of simple proof obligations.

2.2.2 Transformation rules

As with propagation rules, the automatic proof strategy used in the Rhodium system requires an automatic theorem prover to discharge a local soundness property for each transformation rule. Intuitively, a transformation rule is sound if the original and the transformed statements have the same behavior, assuming that the meanings of all incoming facts hold. This property is stated informally in the following definition of soundness for a transformation rule.

Def 5 *A transformation rule* **if** ψ **then transform to** n' *is said to be sound iff it satisfies the following property:*

> *For all concrete execution states η and CFG nodes n, if (1) the rule fires at node n, (2) the meanings of all facts flowing into n hold for η, and (3) the execution of n from η yields η', then the execution of n' from η also yields η'.* (trans-sound)

As an example, consider the transformation rule on lines 8–9 of Figure 2.3. The soundness checker effectively asks the theorem prover to show that the statements $X := Y$ and $X := C$ have the same behavior, under the assumption that X is equal to C. This follows easily from the system-provided concrete semantics of the Rhodium IL.

The following theorem, which is formalized in Chapters 5 and 6, and proved in Appendices C and D, summarizes the part of the proof of soundness of an optimization O that is performed by hand:

Theorem 3 *If all the propagation rules and transformation rules of a Rhodium optimization O are sound, then O is sound.*

As described earlier, the fact that each propagation rule is sound is sufficient to ensure that the induced analysis $\mathcal{A}$ is sound. This fact, along with the fact that each transformation rule is sound, is sufficient to show that all the suggested transformations can be performed without changing the semantics of any IL procedure. Here again, the key to full automation is to split the proof task into an optimization-independent part, done by a human once and for all, and an optimization-dependent part, discharged by an automatic theorem for each optimization. Indeed, the theorem prover only needs to discharge (prop-sound) and (trans-sound), both of which are local, non-inductive properties whose proofs are easy

to automate. The proof of Theorem 3, on the other hand, would be difficult to fully automate, but since the theorem is optimization-independent, it can be proven by a human once and for all.

2.3 Profitability heuristics

In many optimizations, the condition that specifies when a transformation is *legal* can be separated from the condition that specifies when a transformation is *profitable*. Rhodium provides *profitability edge facts* for implementing profitability decisions. Because they are not meant to be used for justifying soundness, these facts have an implicit meaning of *true*, and as a result, they can always be safely added to the CFG. The Rhodium system can therefore give programmers a lot of freedom in computing these facts. In particular, the Rhodium system allows programmers to write regular compiler passes called *profitability analyses*, which are given a read-only view of the compiler's data structures, except for the ability to add profitability facts to the CFG. In this way, one can for example use standard algorithms to annotate the CFG with facts indicating where the loop heads [7] are, what the loop-nest [7] is, or how many times a variable is accessed inside of a loop – these algorithms do not have to be expressed using propagation rules. Transformation rules can then directly use these facts to select only those transformations that are profitable.

2.3.1 An example: loop-induction-variable strength reduction

To illustrate the use of profitability facts, I show how to write loop-induction-variable strength reduction in Rhodium. The idea of this optimization is that if all definitions of a variable I inside of a loop are increments, and some expression $I * C$ is used in the loop, then we can (1) insert $X := I * C$ before the loop (2) insert $X := X + C$ right after every increment of I in the body of the loop and (3) replace $I * C$ with X in the body of the loop. Consider for instance the code snippet in Figure 2.7(a). The result of performing loop-induction-variable strength reduction is shown in Figure 2.7(b). Subsequent passes can clean up this code even further: a constant propagation pass will transform `x := i * 20` to `x := 0`, and then a dead-assignment elimination pass will remove all the assignments to `i`.

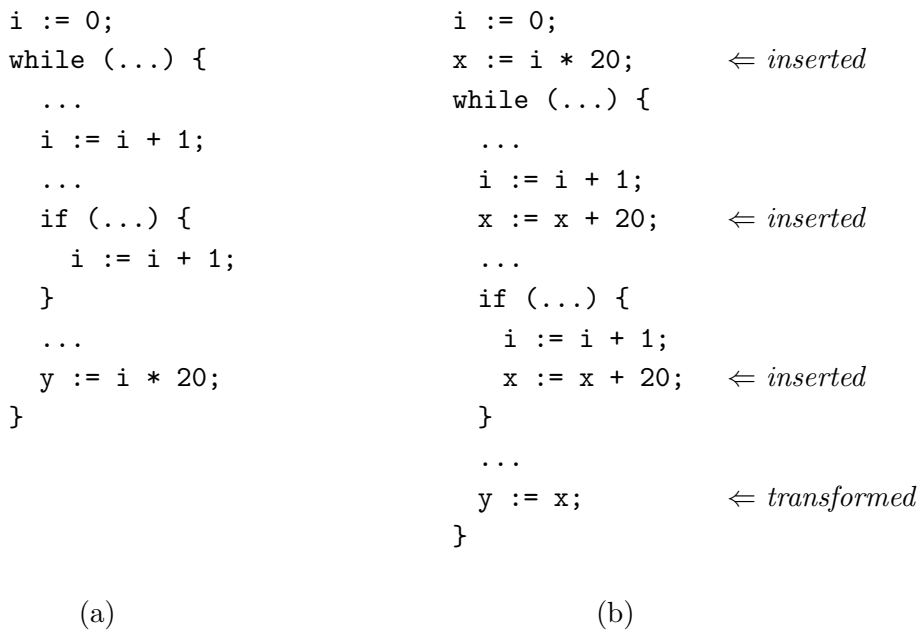


Figure 2.7: Code snippet before and after loop-induction-variable strength reduction

The effect of this optimization can be achieved in Rhodium in two passes. The first pass inserts assignments to the newly created induction variable `x`. The second pass propagates arithmetic invariants to determine that `x = i * 20` holds just before the statement `y := i * 20`, thereby justifying the strength-reduction transformation.

For the first pass, determining when it is safe to insert an assignment is simple: an assignment $X := E$ can be inserted if X is dead after the insertion point, and E does not cause any run-time errors. The tricky part of this first pass lies in determining which of the many legal insertions should be performed so that the later arithmetic-invariant pass can justify the desired strength reduction. This decision of what assignments to insert can be guided by profitability facts. A profitability analysis running standard algorithms can insert the following three profitability facts:

- $indVar(I, X, C)$ is inserted on all the edges in a loop (plus the incoming edge into the loop) to indicate that I is a induction variable in the loop, X is a fresh induction variable that would be profitable to insert, and C is the anticipated multiplication factor between I and X.

- *afterIncr*(I) is inserted on the edge immediately following a statement $I := I + 1$.

- *afterLoopInit*(I) is inserted on the edge immediately following a statement $I := E$ that is at the head of a loop.

In the example of Figure 2.7, *indVar*(`i`, `x`, `20`) would be inserted throughout the loop, *afterIncr*(`i`) would be inserted after the increments of `i` and *afterLoopInit*(`i`) would be inserted after the assignment `i := 0`. The following two transformation rules then indicate which assignments should be inserted:[4]

decl X:*Var*, I:*Var*, C:*Const*

if *stmt*(`skip`) $\wedge$ *dead*(X)@*out* $\wedge$
afterIncr(I)@*out* $\wedge$ *indVar*(I, X, C)@*out*
then transform to $X := X + C$

if *stmt*(`skip`) $\wedge$ *dead*(X)@*out* $\wedge$
afterLoopInit(I)@*out* $\wedge$ *indVar*(I, X, C)@*out*
then transform to $X := I * C$

Analogously to statement removal, statement insertion is expressed in Rhodium as replacement of a `skip` statement. These `skip` statements are only virtual, and the compiler implicitly inserts an infinite supply of them in between any two nodes in the CFG. The above transformations are sound because of the *dead*(X) fact. The other facts are simply there to guide which dead assignments to insert. Since their meaning is *true* and they are used in a conjunction, they do not have any impact on soundness checking.[5]

For the second pass that runs after the dead assignments have been inserted, we can use an arithmetic simplification optimization, shown in Figure 2.8. This optimization is driven by an arithmetic invariant analysis that keeps track of invariants of the form $E_1 = E_2 * E_3$, represented in Rhodium with the *equalsTimes* fact schema. A few representative rules from this analysis are shown in Figure 2.8. These rules make use of a user-defined node fact

[4]This example is made simpler for explanatory purposes by eliding the profitability facts and transformation rules that would insert the declaration of the newly created induction variable.

[5]The profitability facts used here are all backward facts because the rules are all backward. Details on how to check backward rules for soundness can be found in Chapter 6. However, the intuition of the profitability fact being true, and thus disappearing because it is used in a conjunction, remains the same.

1. **define forward edge fact** $equalsTimes(E_1{:}Expr,\ E_2{:}Expr,\ E_3{:}Expr)$
2. **with meaning** $\eta(E_1) = \eta(E_2) * \eta(E_3)$
3. **decl** $E_1{:}Expr,\ E_2{:}Expr,\ E_3{:}Expr$
4. **decl** $X{:}Var,\ Y{:}Var,\ I{:}Var$
5. **decl** $C{:}Int,\ C_1{:}Int,\ C_2{:}Int,\ C_3{:}Int$
6. **if** $equalsTimes(E_1, E_2, E_3)@in \wedge$
7. $\quad unchanged(E_1) \wedge unchanged(E_2) \wedge unchanged(E_3)$
8. **then** $equalsTimes(E_1, E_2, E_3)@out$
9. **if** $stmt(X := I * C) \wedge X \neq I$
10. **then** $equalsTimes(X, I, C)@out$
11. **if** $stmt(I := I + C_1) \wedge X \neq I \wedge$
12. $\quad equalsTimes(X, I, C_2)@in$
13. **then** $equalsTimes(X, I - C_1, C_2)@out$
14. **if** $stmt(X := X + C_1) \wedge X \neq I \wedge$
15. $\quad equalsTimes(X, I - C_2, C_3)@in \wedge$
16. $\quad C_1 = applyBinaryOp(*, C_2, C_3)$
17. **then** $equalsTimes(X, I, C_3)@out$
18. **if** $stmt(Y := I * C) \wedge equalsTimes(X, I, C)@in$
19. **then transform to** $Y := X$

Figure 2.8: Arithmetic simplification optimization in Rhodium

unchanged(E), which says that the current statement does not cause the value of E to change. The rule on lines 14–17 also makes use of the built-in function *applyBinaryOp*, which evaluates a given a binary operator on two integers. The optimization *per se* is performed by a single transformation rule on lines 18–19, which says that a statement $Y := I * C$ can be transformed to $Y := X$ if we know that $X = I * C$ holds before the statement.

The rules in Figure 2.8 are sufficient to trigger the strength-reduction transformation in Figure 2.7(b). The statement `x := i * 20` establishes the dataflow fact *equalsTimes*(`x`, `i`, `20`). Every sequence of `i := i + 1` followed by `x := x + 20` propagates first *equalsTimes*(`x`, `i-1`, `20`) and then *equalsTimes*(`x`, `i`, `20`). As a result, *equalsTimes*(`x`, `i`, `20`) is propagated to `y := i * 20`, thereby triggering the transformation to `y := x`.

The strength-reduction example presented here illustrates two aspects of Rhodium that allow programmers to express complex optimizations, despite Rhodium's restricted syntax. First, much of the complexity of an optimization can be factored into profitability analyses, on which there are no expressiveness limitations. Second, optimizations that traditionally are expressed as having effects at multiple points in the program, such as various sorts of code motion, can in fact be decomposed into several simpler transformations, each of which can be expressed in Rhodium.

The strength-reduction example illustrates both of these points. Loop-induction-variable strength reduction is a complex code-motion optimization, and yet it can be expressed in Rhodium using simple forward and backward passes with appropriate profitability analyses. This way of factoring complicated optimizations into smaller pieces, and separating the part that affects soundness from the part that doesn't, allows users to write optimizations that are intricate and expressive yet still amenable to automated soundness reasoning.

2.3.2 Tags

To express the strength reduction optimization from the previous section, the programmer must be able to sequence the smaller Rhodium optimizations that together create the net effect of strength reduction: first run the profitability analyses, then do dead assignment

{ *dead_assignment_elimination* }
define backward edge fact $dead(X{:}Var)$
with meaning $\eta_1/X = \eta_2/X$

decl $X{:}Var$, $E{:}Expr$

{ *dead_assignment_elimination* }
if $dead(X)@out \wedge mustNotUse(X)$
then $dead(X)@in$

{ *dead_assignment_elimination* }
if $stmt(X := E) \wedge dead(X)@out$
then transform to `skip`

Figure 2.9: Dead assignment elimination with tags

$$\forall f \in facts(r)\ .\ tags(r) \subseteq tags(\textbf{define edge fact } f(\ldots)) \tag{2.1}$$

$$tags(\textbf{define edge fact } f(\ldots)) \subseteq tags(\textbf{if } \psi \textbf{ then } f(\ldots)@\ldots)) \tag{2.2}$$

Figure 2.10: Consistency and completeness requirements for tag annotations

insertion, then arithmetic simplification, and finally dead assignment elimination.

Rhodium allows a programmer to express such sequencing via *tags*. A tag is a string that the programmer attaches to a fact-schema declaration, a transformation rule or a propagation rule. For example, Figure 2.9 shows some of the declarations and rules from dead assignment elimination, tagged with "dead_assignment_elimination" (the meaning of $dead(X)$ will be explained in Section 6.2.3). The Rhodium execution engine then provides the compiler writer with a function *run_tagged_opts* that, given a set T of tags, runs the Rhodium program that contains only those declarations and rules that have a tag in T. The *run_tagged_opts* function can be seen as providing a simple form of slicing of the Rhodium optimization rules, where the programmer can use tag annotations to specify what fact schemas and rules to include in a given slice. If the slice selected by *run_tagged_opts* contains both forward and backward rules, the forward rules are run first, followed by the backward ones.

Tag annotations must satisfy some consistency requirements, so that slices selected by

run_tagged_opts are well formed. For example, it would not make sense to include a transformation rule in a slice, but not the declarations of fact schemas that are used by the transformation rule. Equation (2.1) from Figure 2.10 shows the consistency requirements for tag annotations, where $tags(s)$ represents the user-provided tags for a rule or declaration s, and $facts(r)$ represents the names of all the facts used in a rule r. Equation (2.1) says that if a rule is annotated with a tag, then so must the declaration of all fact schemas referenced in the rule.

Tag annotations should also satisfy a completeness requirement, so that all the rules required to compute a fact are included in the slice. Equation (2.2) shows the completeness requirements for tag annotations: it says that if a fact-schema declaration is annotated with a tag, then so must all the rules propagating instances of that fact schema. Although the completeness requirement is not necessary for the selected slice to be well-formed, it is nevertheless a useful property to have, since it guarantees that no rules were mistakenly omitted from the slice.

Unfortunately, it can be quite burdensome for the programmer to consistently and completely annotate all the rules and declarations in a well-formed slice. To alleviate this burden, the Rhodium system can infer a consistent and complete set of tags for an entire slice given only the tags for the transformation rules. As a result, the programmer only needs to tag transformation rules, with the remaining tags inferred by the system.

Inferring tags consists of finding, for each rule and declaration, the least set of tags that satisfies the two constraints from Figure 2.10, with the additional constraint that tags on transformation rules must match the user-provided tags. A standard fixed-point algorithm for solving subset constraints can achieve this goal. All tags on transformation rules are initialized to the user-provided values, while the remaining tags are initialized to the empty set. The two constraints from Figure 2.10 are then repeatedly applied until no more changes occur. In particular, for each tag appearing on a rule, the algorithm adds the tag to the declaration of all fact schemas used in the rule's antecedent; and for each tag appearing on a fact-schema declaration, the algorithm adds the tag to all the rules that propagate instances of that fact schema.

Putting all this together, Figure 2.11 shows how to control the sequencing for loop-

void *run_loop_induction_variable_strength_reduction*() {
    run_profitability_analyses();
    run_tagged_opts ({"*dead_assignment_insertion*"});
    run_tagged_opts ({"*arithmetic_simplification*"});
    run_tagged_opts ({"*constant_propagation*"});
    run_tagged_opts ({"*dead_assignment_elimination*"});
}

decl X:*Var*, Y:*Var*, I:*Var*
decl C:*Const*, C_1:*Const*, C_2:*Const*, C_3:*Const*
decl E:*Expr*

{ *dead_assignment_insertion* }
if $stmt(\texttt{skip}) \wedge dead(X)@out \wedge$
  $afterIncr(I)@out \wedge indVar(I, X, C)@out$
then transform to $X := X + C$

{ *dead_assignment_insertion* }
if $stmt(\texttt{skip}) \wedge dead(X)@out \wedge$
  $afterLoopInit(I)@out \wedge indVar(I, X, C)@out$
then transform to $X := I * C$

{ *arithmetic_simplification* }
if $stmt(Y := I * C) \wedge equalsTimes(X, I, C)@in$
then transform to $Y := X$

{ *constant_propagation* }
if $stmt(X := Y * C_2) \wedge hasConstValue(Y, C_1) \wedge$
  $C_3 = applyBinaryOp(*, C_1, C_2)$
then transform to $X := C_3$

{ *dead_assignment_elimination* }
if $stmt(X := E) \wedge dead(X)@out$
then transform to `skip`

Figure 2.11: Sequencing of optimizations for loop-induction-variable strength reduction

induction-variable strength reduction, assuming that the compiler is written in a C-like language. At the top of the figure, the function *run_loop_induction_variable_strength_reduction* is the main entry point of the optimization. This function is written in the C-like language that the compiler is written in, and a call to it must be inserted inside the main optimizing loop of the compiler. The optimization begins by calling *run_profitability_analyses*, a function written by the programmer that runs the required profitability analyses, annotating the CFG with instances of the profitability facts *indVar*, *afterIncr*, and *afterLoopInit*. These profitability analyses are also implemented in the language the compiler is written in, and they make use of a function provided by the Rhodium system, namely *add_profitability_fact*, that allows the programmer to add a given profitability fact to a given CFG edge. After the profitability analyses have run to completion, the strength reduction optimization calls *run_tagged_opts* fours times in order to run the rules for dead assignment insertion, arithmetic simplification, constant propagation, and finally dead assignment elimination. The appropriately tagged transformation rules are shown at the bottom of the figure. For brevity, Figure 2.11 only shows those transformation rules that would fire on the example from Figure 2.7.

Since arithmetic simplification and constant propagation are both forward optimizations, and since they are invoked one after another, it would be possible to run them together with a single call to *run_tagged_opts*:

run_tagged_opts ({"*arithmetic_simplification*", "*constant_propagation*"});

Alternatively, one could add a new tag, "post_assignment_insertion", to the arithmetic simplification and constant propagation rules, as shown below:

{ *arithmetic_simplification, post_assignment_insertion* }
if $stmt(Y := I * C) \wedge equalsTimes(X, I, C)@in$
then transform to $Y := X$

{ *constant_propagation, post_assignment_insertion* }
if $stmt(X := Y * C_2) \wedge hasConstValue(Y, C_1) \wedge$
$C_3 = applyBinaryOp(*, C_1, C_2)$
then transform to $X := C_3$

One could then run arithmetic simplification and constant propagation together by calling *run_tagged_opts* ({"*post_assignment_insertion*"). As this example shows, rules can be annotated with multiple tags, and calling *run_tagged_opts*(T) has the effect of running all the rules that have been annotated with a tag from T (or, alternatively, all the rules r for which $tags(r) \cap T \neq \emptyset$).

2.4 Dynamic semantics extensions

The meaning of dataflow facts presented so far all talked about the concrete program states occurring on edges annotated with the fact. Unfortunately, the natural way to express the meaning of certain dataflow facts is to use a predicate over complete traces of program states rather than single program states.

As a motivating example, consider extending the pointer analysis from Figure 2.4 with heap summaries [24, 97], where each allocation statement S represents all the memory blocks allocated at S. The meaning of $mustNotPointTo(X, S)$, where X is a variable and S is an allocation site, is that X does not point to any of the memory blocks allocated at S. This property, however, cannot be expressed by just looking at the current program state, because there is no way to determine which memory blocks were allocated at site S.

One way to fix this problem would be to enrich Rhodium meanings so that they talk about execution traces. From the execution trace one can easily extract the memory blocks that were allocated at site S (by evaluating, for each statement $S : X := \texttt{new}\ T$ in the trace, the value of X in the successor state). However, to extract this information, one has to use quantifiers that range over indices of unbounded-length traces. Unfortunately, I have found the heuristics used in automatic theorem provers for managing quantifiers to be easily confounded by these kinds of quantified formulas that arise when using unbounded-length traces.

To solve this problem Rhodium allows the program state to be extended with user-defined components called *state extensions*. These components are meant to gather the information from a trace that is relevant for a particular dataflow-fact schema. Instead of referring to the trace, the meaning can then refer to the state extension. For the above heap summary example, the state would be extended with a map describing which heap

locations were allocated at which sites, and the meaning of *mustNotPointTo* could then use this map instead of referring to the trace.

To update the user-defined components of the state, programmers also extend the dynamic semantics of the intermediate language. Because of the way these extensions to the semantics are declared, they are guaranteed to be conservative, meaning that the trace of a program in the original semantics and the corresponding trace in the extended semantics agree on all the components of the program state from the original semantics. As a result, if we preserve the extended semantics using the regular Rhodium proof strategy, we are guaranteed to also preserve the original semantics. User defined state extensions are just a formal tool for proving soundness: they can be erased without having any impact on how analyses or IL programs are executed.

I present state extensions in more detail by showing how they can be used to extend the pointer analysis from Figure 2.4 with heap summaries. To define the meaning of *mustNotPointTo* over summaries, I define an additional component of the program state called *summary_of*, which maps each heap location to the heap summary that represents it. I start by considering allocation site summaries, where the locations created at the same site are summarized together by the node that created them. The declaration of *summary_of* then looks as follows:

type *HeapSummary* = *Node*
define state extension
summary_of : *Loc* → *HeapSummary*

The *summary_of* map gets updated according to the following dynamic semantics extension:[6]

decl X:*Var*, T:*Type*

if *stmt*(X := *new* T)
then (η@*out*).*summary_of* =
(η@*in*).*summary_of*[η@*out*(X) ↦ *currNode*]

[6]The new statement in the Rhodium intermediate language creates a dynamically typed slot, and so the actual syntax for new is X := *new*. I use a typed version of the new statement here, namely X := *new* T, in order to show that the techniques presented here would be flexible enough to also support type-based summaries, as shown in Table 2.1.

Table 2.1: Various kinds of heap summarization strategies achievable by varying the definition of *HeapSummary* and the dynamic semantics extension

	HeapSummary	$\eta@out(X)$ maps to this in the dynamic semantics extension
Allocation-site summaries	*Node*	*currNode*
Type-based summaries	*Type*	T
Variable-based summaries	*Var*	X
Single heap summary	*unit*	()

The terms $\eta@in$ and $\eta@out$ refer respectively to the program states before and after the current statement, while the special term *currNode* refers to the current CFG node (*currStmt* refers to the statement at the current node, and *currNode* refers to the actual node). The rule as a whole says that an allocation site $X := new\ T$ updates the *summary_of* component of the state by making the newly created location, obtained by evaluating X in $\eta@out$, map to the CFG node that was just executed. In all other cases the *summary_of* component implicitly remains unchanged.

One can easily modify the above declarations to achieve other kinds of summaries. In particular, Table 2.1 shows how to modify the *HeapSummary* definition and change what $\eta@out(X)$ maps to in the dynamic semantics extension in order to specify different summarization strategies. The rest of my treatment of heap summaries applies to all of the strategies, except when explicitly stated.

The next step is to define the domain of abstract locations:

type $AbsLoc = Var \mid HeapSummary$

An abstract memory location AL of type *AbsLoc* is either a variable or a heap summary. The intuition is that AL represents a set of concrete memory locations: if AL is a variable, it represents the address of the variable; if AL is a heap summary, it represents the set of summarized heap locations.

I can now modify the *mustNotPointTo* fact schema to take abstract locations, instead of just variables (the meaning is explained below):

define edge fact $mustNotPointTo(AL_1{:}AbsLoc, AL_2{:}AbsLoc)$

with meaning

$$\forall L : Loc\ .\ belongsTo(L, AL_1, \eta) \wedge isLoc(\eta(*L)) \Rightarrow \neg belongsTo(\eta(*L), AL_2, \eta)$$

define $belongsTo(L{:}Loc, AL{:}AbsLoc, \eta{:}State) \triangleq$

$$isVar(AL) \Rightarrow [L = \eta(\&AL)] \wedge isHeapSummary(AL) \Rightarrow [\eta.summary\_of[L] = AL]$$

The meaning of *mustNotPointTo* says that none of the locations belonging to AL_1 points to any of the locations belonging to AL_2. The locations belonging to AL_1 are those locations L for which $belongsTo(L, AL_1, \eta)$ holds. For all these locations L,[7] we look up the memory content of L using $\eta(*L)$. If the memory content $\eta(*L)$ is another location, i.e. a pointer, then we want $\eta(*L)$ to *not* belong to AL_2.

The auxiliary function $belongsTo(L, AL, \eta)$ returns whether or not a location L belongs to an abstract location AL in state η. The definition of *belongsTo* is split into two cases, based on the type of AL. If AL is a variable, then L belongs to AL if L is exactly the address of AL. If AL is a heap summary, then L belongs toAL if $\eta.summary\_of$ maps L to AL.

The Rhodium system provides a variety of built-in primitives for testing the types of values, such as for example *isLoc* and *isVar*. The programmer has the ability to define such primitive functions as well, by using *prim* blocks inside the Rhodium code. A prim block is simply a set of axioms, written in the underlying language of the theorem prover, that gets added to the theorem prover's background knowledge. Rhodium programmers can therefore define primitive functions by providing the appropriate axioms in prim blocks to define their behaviors. In their full generality, prim blocks allow programmers to add arbitrary axioms to the system, which could lead to unsoundness, for example if the programmer asks the

[7]The quantifier $\forall L : Loc$ ranges over an infinite set, but it used in a meaning, not in the antecedent of a rule.

theorem prover to assume exactly the obligation that it is being asked to show.

In the current system, programmers must use prim blocks to implement state extensions and meanings that use state extensions (for example, the meaning above, including the *belongsTo* and *isHeapSummary* functions). In essence, the language for implementing these state extensions and meanings is the first-order logic language of the Simplify [36] theorem prover, rather than the nicer language shown above. The Simplify code in these prim blocks is syntactically more complicated than the above code, but logically it performs the same operations. As future work, I would like to formalize the language shown above and modify the Rhodium soundness checker to parse and translate this language into Simplify code, so that programmers are not required to write prim blocks. This will not only make it easier for programmers to write state extensions and meanings over these extensions, but it will also prevent programmers from compromising soundness through prim blocks.

Now that the *mustNotPointTo* fact schema can take heap summaries as parameters, the rules for the pointer analysis from Figure 2.4 must be modified to take these summaries into account. I only present a few representative examples here. The following rule, which works only for allocation-site summaries, says that after an allocation site $X := new\ T$, X does not point to any heap summary that is different from the current node:

decl $Summary{:}HeapSummary$, $X{:}Var$, $T{:}Type$
if $stmt(X := \texttt{new}\ T) \wedge Summary \neq currNode$
then $mustNotPointTo(X, Summary)@out$

To prove this rule sound, the theorem prover must show that the meaning of $mustNotPointTo(X, Summary)$ holds after $X := \texttt{new}\ T$. Since X is a variable and $Summary$ is a heap summary, the meaning expands to $isLoc(\eta(X)) \Rightarrow \eta.summary\_of[\eta(X)] \neq Summary$. Since the theorem prover knows that $\texttt{new}\ T$ returns a location, it determines that $isLoc(\eta(X))$ holds, and then the remaining obligation is $\eta.summary\_of[\eta(X)] \neq Summary$. To prove this, the theorem prover makes use of the user-defined extension to the dynamic semantics. Indeed, if we let η be the program state right after executing the allocation, then the dynamic semantics extension tells us that $\eta.summary\_of[\eta(X)] = currNode$. In conjunction with $Summary \neq currNode$, this implies $\eta.summary\_of[\eta(X)] \neq Summary$,

which is what needed to be shown.

The above rule for $stmt(X := new\ T)$ works only for allocation-site summaries. Of all the pointer analysis rules, it is the only one that depends on the heap summarization strategy. To modify it for other kinds of heap summaries, the antecedent of the rule should compare *Summary* with the third column of Table 2.1, rather than with *currNode*.

As another example, here is the pointer analysis rule that requires the most complex reasoning from the theorem prover:

decl X:*Var*, Y:*Var*, AL_2:*AbsLoc*

if $stmt(X := *Y) \wedge$
$\quad \forall AL_1 : AbsLoc\ .\ mayPointTo(Y, AL_1)@in \Rightarrow$
$\qquad\qquad mustNotPointTo(AL_1, AL_2)@in$
then $mustNotPointTo(X, AL_2)@out$

In the above rule, *mayPointTo* is defined as before: $mayPointTo(AL_1, AL_2) \triangleq \neg mustNotPointTo(AL_1, AL_2)$. The rule as a whole says that X does not point to any of the locations in AL_2 after $X := *Y$ if for all abstract locations AL_1 that Y may point to, it is the case that none of the locations in AL_1 point to any of the locations in AL_2.

The pointer analysis example in this section has shown how user-defined state extensions can be used as a theoretical device for reducing the complexity of obligations sent to the theorem prover. State extensions allow the programmer to re-express meanings that, at first sight, might seem to require mentioning run-time traces, as meanings that only mention the current program state. Since the automated theorem prover used in the Rhodium system has a much easier time reasoning about states than about traces, this ability to convert trace references into state references allows the Rhodium system to reason about a broader class of optimizations.

2.5 Termination

In addition to determining whether or not a Rhodium optimization is sound, the programmer would like to determine whether or not an optimization *terminates*. One easy way of guaranteeing termination is to make the domains of fact-schema parameters finite for a particular intermediate language program. For example, one could define the *Const* and *Expr*

decl X:*Var*, A:*Var*, B:*Var*, C:*Const*, OP:*BinaryOp*

if $stmt(X := A\ OP\ B) \wedge$
$\quad hasConstValue(A, C_1)@in \wedge$
$\quad hasConstValue(B, C_2)@in \wedge$
$\quad C = applyBinaryOp(OP, C_1, C_2)$
then $hasConstValue(X, C)@out$

Figure 2.12: Additional constant propagation rule

domains so that, rather than representing all possible constants and expressions, they represent only those constants and expressions that appear in the intermediate-language program being analyzed. In this scenario, the powerset lattice over which the analysis runs would be finite. Furthermore, as discussed in Section 2.1.3, the flow function F is guaranteed to be monotonic, and so the dataflow values computed by iterative analysis form an ascending chain. This, combined with the finiteness of the domain, guarantees termination [82].

Although it may be appealing to restrict *Const* and *Expr* to be finite for the sake of termination, the infinite unrestricted versions of *Const* and *Expr* are important for achieving Rhodium's expressive power.[8] For example, being able to refer to expressions that are not in the analyzed program is crucial for expressing the arithmetic invariant analysis *equalsTimes* from Figure 2.8. Furthermore, by making use of infinite domains, Rhodium can perform range analysis where the end points of the range are not restricted to constants in the program. Finally, Rhodium can express a version of constant propagation that constructs and propagates constants that are not in the source code, as shown by the rule in Figure 2.12.

However, with this extra flexibility comes a challenge: whereas analyses over finite domains are trivially guaranteed to terminate, analyses over infinite domains, on the other hand, may very well run forever. There are two ways in which a Rhodium analysis might run forever. The first one is that a particular rule might not terminate. The second is that the fixed-point computation might not terminate. I deal with each one of these in the next

[8] When I say finite or infinite here, I mean finite or infinite *once* a given finite intermediate-language program has been chosen.

two subsections.

2.5.1 Termination of a single rule

To guarantee that execution of a rule terminates, one must guarantee that the rule has only a finite number of instantiations (i.e., substitutions for its free variables), and that each instantiation can be evaluated in finite time. For the latter, I restrict the logic of each rule's antecedent to the decidable subset of first-order logic in which quantifiers range only over finite domains.[9]

For the former, each rule must intuitively satisfy a "finite-in-finite-out" property: if a rule is invoked on a node where all incoming edges have finite sets of facts, then the rule will have only a finite number of instantiations and will generate only a finite set of facts on outgoing edges. Unfortunately, unrestricted propagation rules do not have that property: it is possible for a sound rule to propagate infinitely many dataflow facts, even when the input facts are finite. For example, consider the following sound range-analysis rule:

> **define forward edge fact** $inRange(X : Var, lo : Const, hi : Const)$
> **with meaning** $lo \leq \eta(X) \wedge \eta(X) \leq hi$
>
> **if** $stmt(X := C) \wedge C_1 \leq C \wedge C_2 \geq C$
> **then** $inRange(X, C_1, C_2)@out$

There are infinitely many instantiations of C_1 and C_2 that will make this rule fire, even if the input contains no dataflow facts.

To prevent such a situation, I adapt a notion from the database community called *safety* [116]. A Rhodium propagation rule is said to be *finite-safe* if every free variable of infinite domain in the consequent is finite-safe. A variable is finite-safe if it appears (after expanding virtual facts and folding away all negations) in the antecedent either in a dataflow fact, or on one side of an equality where the other side contains only finite-safe variables; finite-safe variables thus are constrained to have a finite number of instantiations if the input fact set is finite. The range-analysis rule above is not finite-safe, since neither C_1 nor C_2 is finite-safe.

---

[9] Here again, the domain must be finite for a *particular* program, not necessarily for all programs.

Even if all rules are finite-safe, a rule can still be invoked on an infinite input set: $\perp$. This case can happen at the start of analysis, since all edges (aside from the entry edge) are initialized with $\perp$. However, $\perp$ can be treated specially without invoking any of the user-defined propagation rules. In particular, for nodes that have one input edge, it is sound to propagate $\perp$ when the input is $\perp$. For nodes that have two input edges (merge nodes), if any one of the two inputs is $\perp$, then it is sound to propagate the other input (whether it be $\perp$ or not).

Thus, if all rules are finite-safe, either they will be invoked with $\perp$ on all input edges, and immediately propagate $\perp$, or they will be invoked with a finite set of input facts on some edge and propagate another finite set of output facts in finite time.

2.5.2 Termination of the fixed-point computation

As discussed in Section 2.1.3, the flow function F is guaranteed to be monotonic, and so the dataflow values computed by iterative analysis form an ascending chain. To guarantee termination, all that is left is to ensure that all ascending chains in the lattice have finite length.

In order to do this, recall from Section 2.5.1 that the *finite-safe* requirement was imposed on all propagation rules, which led to all propagated sets being either finite or $\perp$. It is therefore possible to shrink the lattice of the analysis to include only these finite sets and $\perp$. The original underlying lattice was the power-set lattice, in which the ordering was the superset relation. The shrunken lattice uses this same ordering, which means that all *ascending* chains in the shrunken lattice must have a finite length, since the longest chain of *decreasing-sized* finite sets is finite. Notice that the lattice does not have a finite height, because there can still be infinite *descending* chains.

The technique presented here for guaranteeing termination is effective even in the face of fact schemas with infinite-domain parameters. For example, the *equalsTimes* fact schema has all three of its parameters ranging over infinite domains, and yet the Rhodium system is still able to guarantee that the analysis terminates. In this case, the shrunken lattice is infinitely wide and infinitely tall, but its ascending chains are nonetheless guaranteed to be finite.

2.5.3 Custom merges

The range-analysis propagation rule in Section 2.5.1 was sound but not finite-safe: it could produce an infinite (and non-$\perp$) set of output *inRange* facts. However, the meaning of one of the propagated *inRange* facts, $inRange(X, C, C)$, implies all the others' meanings. So an alternative sound and finite-safe propagation rule could be the following:

if $stmt(X := C)$
then $inRange(X, C, C)@out$

Unfortunately, this propagation rule interacts poorly with the powerset lattice's join function, intersection. If intersection is used to join the fact set $\{inRange(\mathtt{x}, 1, 1)\}$ with $\{inRange(\mathtt{x}, 2, 2)\}$, the resulting set is $\{\}$. One would prefer instead to get the fact set $\{inRange(\mathtt{x}, 1, 2)\}$: this fact set is sound (and precise) since its meaning is exactly the disjunction of the meanings of the two merging fact sets.

Rhodium avoids this information-loss problem while retaining finite-safe propagation rules by allowing programmers to define their own merges. Rather than provide special syntax for defining merge functions, I simply introduce a `merge` statement for which users can write ordinary Rhodium propagation rules:

decl X:*Var*, C_1:*Int*, C_2:*Int*, C_3:*Int*, C_4:*Int*

if $stmt(\mathtt{merge}) \wedge$
  $inRange(X, C_1, C_2)@in[0] \wedge$
  $inRange(X, C_3, C_4)@in[1]$
then $inRange(X, min(C_1, C_3), max(C_2, C_4))@out$

This rule, which uses edge indices as described in Section 2.1.5, propagates the union of its two incoming ranges. The functions *min* and *max* are provided as primitives by the Rhodium system, but the programmer could have defined these as well using prim blocks.

When a rule refers to multiple input or output edges, there is one proof obligation sent to the theorem prover for each input-output-edge pair. The general version of (prop-sound) that handles an arbitrary number of input and output edges is given in Chapters 5 and 6. In the above case, there would be two proof obligations, one for input edge 0 and one for input edge 1. For input edge 0, the soundness checker would ask the theorem prover to show that

if the meaning of $inRange(X, C_1, C_2)$ holds of some program state η, and η on edge 0 steps to η' through the merge node, then the meaning of $inRange(X, min(C_1, C_3), max(C_2, C_4))$ holds of η'. A similar proof obligation would be generated for input edge 1.

From a formal point of view, the lattice of the implicitly defined dataflow analysis $\mathcal{A}$ must be modified in order to take into account custom merge functions. Consider the example above, where the merge of $S = \{inRange(\mathtt{x}, 1, 1)\}$ and $T = \{inRange(\mathtt{x}, 2, 2)\}$ gives $merge(S, T) = \{inRange(\mathtt{x}, 1, 2)\}$. In the powerset lattice of all dataflow facts, the expressions S, T and $merge(S, T)$ are unrelated. To prove the soundness of the custom merge, the lattice must be defined in such a way so that $S \sqcup T \sqsubseteq merge(S, T)$ holds, meaning that the user's merge function returns an approximation of the best possible merge (which is $\sqcup$).

To address this problem, when a user-defined merge function is specified, the implicitly defined analysis $\mathcal{A}$ must run over a more general lattice, namely the lattice of predicates: $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot) = (Pred, \vee, \wedge, \Rightarrow, true, false)$. This lattice subsumes the powerset lattice since a set of dataflow facts can be interpreted as a predicate by taking the conjunction of the meanings of all the dataflow facts in the set. The view shown to the programmer is still that sets of dataflow facts are being stored on edges, but from a formal point of view, these sets are interpreted as predicates. In the example above, S becomes $1 \leq \mathtt{x} \leq 1$, T becomes $2 \leq \mathtt{x} \leq 2$, and $merge(S, T)$ becomes $1 \leq \mathtt{x} \leq 2$. Therefore $S \sqcup T = (1 \leq \mathtt{x} \leq 1) \vee (2 \leq \mathtt{x} \leq 2)$, and since $(1 \leq \mathtt{x} \leq 1) \vee (2 \leq \mathtt{x} \leq 2) \Rightarrow 1 \leq \mathtt{x} \leq 2$, we now have $S \sqcup T \sqsubseteq merge(S, T)$ as desired. More generally, if a `merge` rule passes property (prop-sound), then it is guaranteed that if S and T are the two incoming predicates to the `merge` node, then the outgoing predicate $merge(S, T)$ will satisfy $S \vee T \Rightarrow merge(S, T)$, or $S \sqcup T \sqsubseteq merge(S, T)$ in the lattice of predicates.

Unfortunately, the lattice of predicates, even when shrunken to the meanings of only finite sets of facts plus $\bot$, does not have the finite-ascending-chain property. Consider, for example, the *inRange* fact schema, and the infinite sequence $S_0, S_1, S_2, \ldots$, where $S_i = \{inRange(\mathtt{x}, 0, i)\}$. Each one of the sets S_i is finite and therefore belongs to the shrunken lattice; furthermore the sequence is an ascending chain, because each S_i implies S_{i+1}. Consequently, termination of the fixed-point computation is not guaranteed of anal-

yses using custom merges, and indeed the kind of range analysis discussed here does not terminate.[10]

To allow the optimization writer to achieve termination in such cases, as well as allowing the optimization writer to make terminating analyses converge faster, Rhodium provides widening operators [29]. A Rhodium widening operator is a function, written in the underlying language of the compiler, that takes a node, an incoming dataflow fact set, and an "unwidened" outgoing dataflow fact set, and produces the widened outgoing fact set. The *run_tagged_opts* function described in Section 2.3.2 takes as an additional parameter one of these widening operators. After the Rhodium evaluation engine runs the propagation rules on a node n, given an input set d_{in} to produce an "unwidened" output set d_{out}, the widening operator, which was provided as a parameter to *run_tagged_opts*, is run on n, d_{in}, and d_{out} to produce the widened output set d_{wide}. Finally, the engine computes $merge(d_{out}, d_{wide})$ (using either the default merge or a custom merge if one is specified) as the final outgoing set to propagate. From the soundness of F we know that the facts d_{out} are sound, and since $merge(d_{out}, d_{wide})$ is more conservative than d_{out}, $merge(d_{out}, d_{wide})$ must also be sound, regardless of the value of d_{wide}. This means that the value d_{wide} returned by the widening operator does not affect soundness – it only makes the result more conservative, thus helping the iterative analysis reach a fixed point faster. The Rhodium system cannot, however, statically guarantee that the widening operator provided by the programmer is strong enough to ensure termination.

[10] Or, if using bounded-sized integers, it takes a long time.

Chapter 3

PROGRAMS MANIPULATED BY RHODIUM OPTIMIZATIONS

This chapter describes the programs over which Rhodium optimizations run. I start by describing the Rhodium intermediate language (IL), which is a textual representation of these programs. I then present a CFG-based intermediate representation (IR) for these IL programs. Finally, the chapter concludes with the operational semantics of this IR.

This chapter serves mostly as documentation of the current Rhodium IL and its semantics, so that Rhodium programmers can understand how to write rules that are sound. The details of the Rhodium IL are in fact quite orthogonal to the main contribution of this dissertation. Once the semantics of the Rhodium IL is defined in Section 3.4 using a step relation, the rest of the dissertation treats this step relation as a black box. The only place where the step relation comes up again is in the background axioms that the theorem prover uses to discharge soundness obligations (see Section 5.4.4). As a result, many of the details of the syntax and semantics of the Rhodium IL can be changed with little effort.

3.1 The intermediate language

Rhodium optimizations run over a C-like IL with functions, recursion, pointers to dynamically allocated memory and to local variables, and arrays. Figure 3.1 shows the grammar of IL programs. An IL program is a set of procedures, where each procedure contains a sequence of statements. A procedure takes one parameter, and returns one value using the `return` statement. The Rhodium IL is dynamically typed, and so procedure declarations do not include type annotations. Similarly, declarations of IL variables do not include types.

An IL program does not have a distinguished entry procedure – it may be entered by calling any of its procedures. The Rhodium system preserves the semantics of each procedure in isolation, and so IL programs can be compiled separately and then safely linked together.

Statements or expressions in the IL can get stuck, meaning that they cannot be evaluated

ILProg	::=	*Proc* ... *Proc*	
Proc	::=	*ProcName*(*ILVar*) {*Stmt*; ... ; *Stmt*;}	
Stmt	::=	`decl` *ILVar*	(variable declaration)
	|	`decl` *ILVar*[*ILVar*]	(inline array declaration)
	|	`skip`	(no-op statement)
	|	*ILVar* := `new`	(dynamically allocated variable)
	|	*ILVar* := `new array`[*ILVar*]	(dynamically allocated array)
	|	*ILVar* := *Expr*	(simple assignment)
	|	∗*ILVar* := *ILVar*	(pointer store)
	|	*ILVar* := *ProcName*(*BaseExpr*)	(function call)
	|	`if` *BaseExpr* `goto` *Label* `else` *Label*	(branch)
	|	`label` *Label*	(label declarations)
	|	`return` *ILVar*	(return)
Expr	::=	*BaseExpr*	
	|	`*`*ILVar*	
	|	`&`*ILVar*	
	|	*ILVar*[*ILVar*]	
	|	*OP*(*BaseExpr*, ..., *BaseExpr*)	
BaseExpr	::=	*ILVar*	
	|	*Constant*	
Constant	::=	`false` | `true` | `0` | `1` | `2` | ...	
ILVar	::=	‘*id*	(IL variable names)
ProcName	::=	‘*id*	(procedure names)
Label	::=	‘*id*	(label identifiers)
OP	::=	various operators with arity ≥ 1	
id	::=	identifiers	

Figure 3.1: Rhodium intermediate language

any further. If a statement or expression tries to violate type-safety or memory-safely, for example by dereferencing an integer, then it will get stuck. Statements can also get stuck if they violate basic typechecking requirements, for example, by trying to call a procedure that does not exist, or referencing a variable that has not been declared. One interpretation of stuckness is that it represents run-time errors. Another interpretation, which is the one taken in this dissertation, is that stuckness is the result of an underspecified semantics: when a statement is stuck in a program state, then its semantics in that program state is simply unspecified. Section 4.2 discusses how stuckness interacts with the requirement of preserving the semantics of programs.

The following descriptions give an informal semantics of the IL statement forms, including when they get stuck. The full details are in the formal description of the small-step semantics in Section 3.4.

- `decl` `‘x` declares a single dynamically-typed variable `x`, and initializes it to the special value *uninit*. This special value is used in the definition of the IL semantics to represent uninitialized memory locations, and it is not available in the syntax of IL programs. If the variable `x` has already been declared, the statement is stuck. IL variable names (as well as procedure names and label names) are preceded by a ‘ mark. Although this seems unnecessary at this point, the need for a ‘ mark will become clear once the syntax of the Rhodium IL is embedded in the syntax of Rhodium. At that point, the syntax will include both Rhodium variable names, and IL variable names. The ‘ mark will distinguish the two kinds of names: names starting with a ‘ will be IL variables, and names without a ‘ will be Rhodium variables. To make code snippets easier to read in this dissertation, I use fonts and typecase instead of a ‘ mark to distinguish between the two kinds of names: lower case names in `typewriter` font, for example `x`, will refer to Rhodium IL variables, and capital names in *ITALICS* font, for example X, will refer to Rhodium variables.

- `decl` `a`[`i`] declares an inline (stack-allocated) array with `i` dynamically-typed elements, each one initialized with *uninit*. If `a` has already been declared, or `i` does not contain an integer greater than 0, then the statement is stuck.

- `skip` is a no-op, and it is never stuck.

- `x` := `new` allocates a single dynamically-typed slot on the heap, initializes its contents to *uninit*, and stores the address of the slot in `x`. If `x` is not declared, the statement is stuck.

- `x` := `new array`[`i`] allocates an array on the heap with `i` dynamically-typed elements, each one initialized with *uninit*. If `x` is not declared, or `i` is not an integer greater than 0, then the statement is stuck.

- `x` := *Expr* is an assignment to a variable `x`. An expression *Expr* is either a base expression (which is a constant or a variable), a variable dereference, a variable reference (address-of operator), an array index operation, or an arithmetic operator OP. If `x` is not declared, or *Expr* gets stuck, then the statement is stuck. Furthermore, the assignment is also stuck if it attempts to copy an aggregate value, for example a whole array.

- $*$`x` := `y` is a pointer store. If `x` or `y` are not declared, or `x` does not point to a valid location, the statement is stuck. The statement is also stuck if it attempts to copy an aggregate value.

- `x` := `f`(*BaseExpr*) is a function call. If either `x` is not declared, the function `f` does not exist, *BaseExpr* gets stuck, or *BaseExpr* is an aggregate value, then the statement is stuck. Furthermore, the call is stuck if it would cause a stack overflow.

- `if` *BaseExpr* `goto` $\texttt{l}_1$ `else` $\texttt{l}_2$ is a branch statement. If *BaseExpr* evaluates to 1 (`true`), the branch goes to $\texttt{l}_1$; if *BaseExpr* evaluates to 0 (`false`), the branch goes to $\texttt{l}_2$; otherwise, the statement is stuck.

- `label l` is a label statement. It acts as a no-op, and it is never stuck.

- `return x` is a return statement. If `x` is not declared, or it contains an aggregate value, the statement is stuck.

$$
\begin{array}{rcll}
\mathit{Stmt} & ::= & \ldots & \\
& | & *((\&\mathit{ILVar})[\mathit{ILVar}]) := \mathit{ILVar} & \text{(write to inline array)} \\
& | & \mathit{ILVar} := *((\&\mathit{ILVar})[\mathit{ILVar}]) & \text{(read from inline array)} \\
& | & *(\mathit{ILVar}[\mathit{ILVar}]) := \mathit{ILVar} & \text{(write to heap array)} \\
& | & \mathit{ILVar} := *(\mathit{ILVar}[\mathit{ILVar}]) & \text{(read from heap array)}
\end{array}
$$

Figure 3.2: Additional statement forms for convenient array access

The Rhodium IL contains most of the features of C. The main features from C that are missing are structures and pointer arithmetic. However, these omitted features are similar to ones already handled by the Rhodium system. In particular, because Rhodium arrays are heterogeneous, structures can be modeled exactly like arrays, except that values are indexed by strings, rather than integers. Furthermore, pointer arithmetic could be modeled in the same way as array indexing. As a result, the ideas presented in this dissertation should be easily adaptable to a language with structures and pointer arithmetic.

The Rhodium IL also differs from C in the way that arrays are accessed. The $\cdot[\cdot]$ operator in the Rhodium IL returns the address of an array element, rather than the element itself. In particular, the expression $x[i]$ takes a pointer x to some array and an index i, and returns the address of the i^{th} element of the array. After getting a pointer to a array element using $\cdot[\cdot]$, the actual element can be accessed using a regular dereference. As a simple example, the following two IL code snippets both allocate an array of size 10, and then store 5 in the 0^{th} element (the code on the left allocates the array on the heap, while the code on the right allocates the array on the stack):

```
decl i;
i := 10;
a := new array[i];
decl tmp;
tmp := a[0];
*tmp := 5;
```

```
decl i;
i := 10;
decl a[i];
decl addra;
addra := &a;
decl tmp;
tmp := addra[0];
*tmp := 5;
```

For convenience, the Rhodium IL provides additional statement forms, shown in Figure 3.2, for writing and reading array elements directly. These "compound" statement forms are redundant, in that they can be desugared into statements from the simpler language of Figure 3.1.

Finally, the Rhodium IL does not have explicit I/O instructions. Instead, I/O is modeled by having an I/O library that IL programs call into. From the point of view of a client IL program, functions in the I/O library are black boxes. The Rhodium system can already handle calls to functions for which it does not have access to the body, and so it can handle calls to I/O functions in exactly the same way.

3.2 Notation

Before presenting the intermediate representation for IL programs in Section 3.3, this section describes some of the notation used in the remainder of this dissertation. If A is a set, then A^* is the set of all tuples of elements of A. Formally, $A^* \triangleq \bigcup_{i \geq 0} A^i$, where $A^k = \{(a_0, \ldots, a_{k-1}) | a_i \in A\}$. I denote the i^{th} projection of a tuple $x = (x_0, \ldots, x_{k-1})$ by $x[i] \triangleq x_i$. Given a function $f : A \to B$, I extend f to work over tuples by defining $\overrightarrow{f} : A^* \to B^*$ as $\overrightarrow{f}((x_0, \ldots, x_k)) \triangleq (f(x_0), \ldots, f(x_k))$. I also extend f to work over maps by defining $\widetilde{f} : (O \to A) \to (O \to B)$ as $\widetilde{f}(m) \triangleq \lambda o.f(m(o))$.

I extend a binary relation $R \subseteq 2^{D \times D}$ over D to tuples by defining the $\overrightarrow{R}$ relation by: $\overrightarrow{R}((x_0, \ldots, x_k), (y_0, \ldots, y_k))$ iff $R(x_0, y_0) \wedge \ldots \wedge R(x_k, y_k)$. Finally, I extend a binary relation $R \subseteq 2^{D \times D}$ to maps by defining the $\widetilde{R}$ relation as: $\widetilde{R}(m_1, m_2)$ iff for all elements o in the domain of both m_1 and m_2, it is the case that $R(m_1(o), m_2(o))$. To make the equations clearer, I drop the tilde and arrow annotations on binary relations when they are clear from context.

A graph is defined as a tuple $g = (N, E, in, out, InEdges, OutEdges)$ where $N \subseteq Node$ is a set of nodes (with $Node$ being a predefined infinite set), $E \subseteq Edge$ is a set of edges (with $Edges$ being a predefined infinite set), $in : N \to E^*$ specifies the input edges for a node, $out : N \to E^*$ specifies the output edges for a node, $InEdges \in E^*$ specifies the input edges of the graph, and $OutEdges \in E^*$ specifies the output edges of the graph. I denote by $Graph$ the set of all graphs. When necessary, I use subscripts to extract the components

of a graph. For example, if g is a graph, then its nodes are N_g, its edges are E_g, and so on.

For each edge in a graph, there is at most one node that is the source of the edge, and at most one node that is its destination. Given a graph g and an edge $e \in E_g$, $src_g(e)$ and $dst_g(e)$ return the source and destination nodes of edge e, if they exist. In particular:

$$src_g(e) = n \text{ iff } \exists i.out_g(n)[i] = e$$
$$dst_g(e) = n \text{ iff } \exists i.in_g(n)[i] = e$$

Because each edge has at most one source and one destination, there can be at most one i that satisfies each of the above equations.

Given a graph g, an edge $e \in E_g$, and a node $n \in N_g$, $inIndex_g(e, n)$ returns the index in the array $in_g(n)$ that e occupies, and $outIndex_g(e, n)$ returns the index in the array $out_g(n)$ that e occupies. In particular:

$$inIndex_g(e, n) = i \text{ iff } in_g(n)[i] = e$$
$$outIndex_g(e, n) = i \text{ iff } out_g(n)[i] = e$$

Here again, because each edge has at most one source and one destination, there can be at most one i that satisfies each of the above equations. Note that *src*, *dst*, *inIndex*, and *outIndex* are all partial functions.

3.3 The intermediate representation

An IL program is represented as a set of procedures, where each procedure is represented as a control flow graph (CFG), and each node in the CFG represents a statement from the intermediate language. The CFG is a graph, as defined in Section 3.2. In the case of a forward analysis, the CFG will be in the forward direction, and in the case of a backward analysis, the CFG will be in the backward direction, meaning that the direction of the edges is reversed. Label statements do not appear in the IR – once they have been used to build the CFG, they are thrown away. Also, there is one additional statement form that can appear in the IR, but which was not in the grammar of the intermediate language: `merge` statements, which are used to represent merge points in the CFG.

Formally, a program is a tuple $\pi = (p_1, \ldots, p_n)$, where each p_i is a procedure, and a procedure is a tuple $p = (name, formal, cfg)$, where *name* is the name of the procedure, *formal*

is the name of its formal parameter, and *cfg* is its CFG. Let $Prog$ be the set of all programs, and $Proc$ the set of all procedures. I sometimes use the sugar $proc(name, formal, cfg)$ to denote the procedure $(name, formal, cfg)$. I also denote by $name(p)$, $formal(p)$ and $cfg(p)$ the name, formal name, and CFG of a procedure p. I assume that all procedures in a program have distinct names.

Each node n in the representation of a program has a statement associated with it, which is denoted by $stmtAt(n)$. Furthermore, each node n belongs to exactly one program, which is denoted by $prog(n)$. Edges, on the other hand, can be shared between programs, meaning that an edge can appear in multiple programs.

Given a node n for which $stmtAt(n) = (x := f(b))$, I assume that there is a procedure p in $prog(n)$ for which $name(p) = f$. I use $callee(n)$ to denote this procedure p.

The CFGs discussed thus far are intraprocedural, meaning that the successor of a call node is the first node in the caller that gets executed after the callee returns. The Rhodium intermediate representation for a program π also contains an interprocedural CFG, denoted by $CFG(\pi)$, which is derived from the intraprocedural ones. The interprocedural CFG is a graph that connects the intraprocedural CFGs together. Each call node is split into two nodes in the interprocedural CFG: a call node and a return site node. Figure 3.3 shows the intraprocedural CFGs for a program with three procedures, and Figure 3.4 shows the corresponding interprocedural CFG. The node n_1 in Figure 3.3 gets split into nodes n_2 and n_3 in Figure 3.4. Given a call node n in the intraprocedural CFG, I use $interCallNode(n)$ to denote the call node in the interprocedural CFG corresponding to n. In the above example, this would mean that $interCallNode(n_1) = n_2$. Also, given a node n, and assuming $\pi = prog(n)$, I use $in(n)$ and $out(n)$ to denote $in_{CFG(\pi)}(n)$ and $out_{CFG(\pi)}(n)$.

3.4 Small-step semantics

In defining the small-step semantics of the Rhodium IR, I assume an infinite set *Location* of memory locations, with metavariable l ranging over the set. I assume that the set *Const* is disjoint from *Location* and contains integers and the distinguished element *uninit*. I denote by $\mathbb{Z} \subseteq Const$ the set of integers, and by $\mathbb{N} \subseteq \mathbb{Z}$ the set of natural numbers, which are integers greater or equal to 0. I also assume a set *Array* of array values, disjoint from *Const*

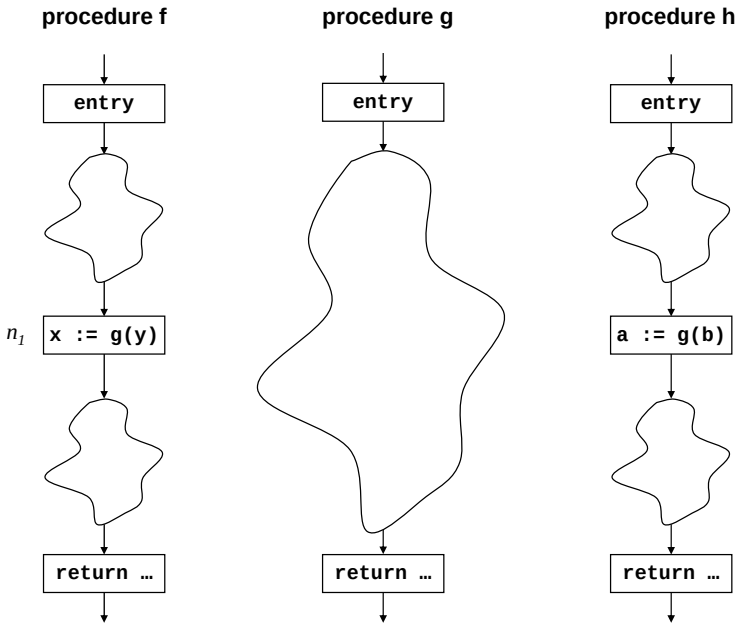


Figure 3.3: Example of intraprocedural CFGs for a program with three procedures

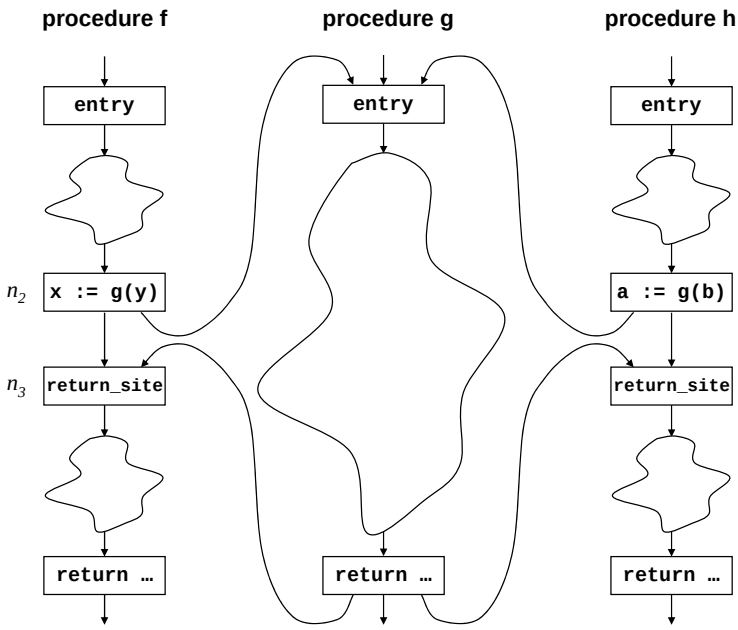


Figure 3.4: The interprocedural CFG for the program in Figure 3.3

and *Location*. An array value is a pair $(len, locs)$, where $len \in \mathbb{N}$ is the length of the array and $locs \in (\mathbb{N} \rightharpoonup \mathit{Location})$ is a map from indices to locations. The locations mapped to are the locations of the array elements. Given an array a, I use $len(a)$ to denote its length, and $locs(a)$ to denote the addresses of its contents. I use $newInitArray(l_0, \ldots, l_j)$ to denote the array $(j+1, \lambda i.l_i)$.

The set of values is defined as $\mathit{Value} = (\mathit{Location} \cup \mathit{Const} \cup \mathit{Array})$.

An *environment* is a partial function $\rho : \mathit{Var} \rightharpoonup \mathit{Location}$, where *Var* is the set of variables in the Rhodium IR; I denote by *Environment* the set of all environments. A *store* is a partial function $\sigma : \mathit{Location} \rightharpoonup \mathit{Value}$; I denote by *Store* the set of all stores. The domain of an environment ρ is denoted $dom(\rho)$, and similarly for the domain of a store. If $s = (x_1, \ldots x_n)$, $s \in dom(\rho)$ denotes that each element of s is in $dom(\rho)$; similar notation is defined for a store σ. The notation $\rho[x \mapsto l]$ denotes the environment identical to ρ but with variable x mapping to location l; if $x \in dom(\rho)$, the old mapping for x is shadowed by the new one. The notation $\sigma[l \mapsto v]$ is defined similarly. The notation $\sigma[l_1 \mapsto v_1, \ldots, l_n \mapsto v_n]$ denotes $\sigma[l_1 \mapsto v_1] \ldots [l_n \mapsto v_n]$. I use $(l_1, \ldots, l_n) \mapsto v$ to stand for $l_1 \mapsto v, \ldots, l_n \mapsto v$. Finally, the notation $\sigma / \{l_1, \ldots, l_i\}$ denotes the store identical to σ except that all pairs $(l, v) \in \sigma$ such that $l \in \{l_1, \ldots, l_i\}$ are removed.

The current dynamic call chain is represented by a stack. A *stack frame* is a triple $f = (n, l, \rho) : \mathit{Node} \times \mathit{Location} \times \mathit{Environment}$. Here n is the CFG node that made the call to the function currently being executed, l is the location in which to put the return value from the call, and ρ is the current lexical environment at the point of the call. I denote by *Frame* the set of all stack frames. A *stack* $\xi = (f_1, \ldots, f_i) : \mathit{Frame}^*$ is a sequence of stack frames, representing the contents of the stack. I assume a global constant $\mathit{maxStackDepth} \in \mathbb{N}$ that represents the maximum size of the stack. Given a stack $\xi = (f_1, \ldots, f_i)$, it is therefore the case that $0 \leq i \leq \mathit{maxStackDepth}$, with ξ being the empty stack $()$ when $i = 0$. The *maxStackDepth* constant is arbitrary, and as a result the Rhodium system can preserve the semantics of programs using arbitrarily large stacks. The set of all stacks is denoted by

Stack. Stacks support two operations defined as follows:

$$
\begin{aligned}
&\mathit{push} : (\mathit{Frame} \times \mathit{Stack}) \rightharpoonup \mathit{Stack} \\
&\mathit{push}(f, (f_1, \ldots, f_i)) = (f, f_1, \ldots, f_i), \text{where } 0 \leq i < \mathit{maxStackDepth} \\
&\\
&\mathit{pop} : \mathit{Stack} \rightharpoonup (\mathit{Frame} \times \mathit{Stack}) \\
&\mathit{pop}((f, f_1, \ldots, f_i)) = (f, (f_1, \ldots, f_i)), \text{where } 0 \leq i < \mathit{maxStackDepth}
\end{aligned}
$$

Note that these two operations are partial: *push* is not defined in the case of a stack overflow, and *pop* is not defined in the case of a stack underflow.

Finally, a memory allocator $\mathcal{M}$ is an infinite stream $\langle l_1, l_2, \ldots \rangle$ of distinct locations. I denote the set of all memory allocators as *MemAlloc*.

A *state* of execution of a program π is a four-tuple $\eta = (\rho, \sigma, \xi, \mathcal{M})$ where $\rho \in \mathit{Environment}$, $\sigma \in \mathit{Store}$, $\xi \in \mathit{Stack}$, and $\mathcal{M} \in \mathit{MemAlloc}$. I denote the set of program states by *State*. I refer to the environment of a state η as $\mathit{env}(\eta)$, and I similarly define accessors *store*, *stack*, and *mem*.

In the following definition of the expression evaluation function, the arity of an operator *op* is denoted $\mathit{arity}(op)$, and I assume, for each n-ary operator symbol *op*, a fixed interpretation function $[\![op]\!] : \mathit{Const}^n \rightharpoonup \mathit{Const}$.

Definition 1 *The evaluation of an expression e in a program state η, where $\mathit{env}(\eta) = \rho$ and $\mathit{store}(\eta) = \sigma$, is given by the partial function $\eta(e) : (\mathit{State} \times \mathit{Expr}) \rightharpoonup \mathit{Value}$ defined by:*

$$
\begin{aligned}
\eta(c) &= c \\
\eta(\&x) &= \rho(x) \\
&\quad \mathit{where}\ x \in \mathit{dom}(\rho) \\
\eta(x) &= \sigma(\rho(x)) \\
&\quad \mathit{where}\ x \in \mathit{dom}(\rho), \rho(x) \in \mathit{dom}(\sigma) \\
\eta(*x) &= \sigma(\eta(x)) \\
&\quad \mathit{where}\ \eta(x) \in \mathit{dom}(\sigma) \\
\eta(op\ b_1 \ldots b_n) &= [\![op]\!](\eta(b_1), \ldots, \eta(b_n)) \\
&\quad \mathit{where}\ \mathit{arity}(op) = n\ \mathit{and}\ \forall\ 1 \leq j \leq n\ .\ (\eta(b_j) \in \mathit{Const}) \\
\eta(x[i]) &= \mathit{locs}(\eta(*x))(\eta(i)) \\
&\quad \mathit{where}\ \eta(*x) \in \mathit{Array}, \eta(i) \in \mathbb{N}, 0 \leq \eta(i) < \mathit{len}(\eta(*x))
\end{aligned}
$$

Note that $\eta(e)$ is partial because of the side conditions in the above definition.

Each node n in the CFG has a set of input edges and a set of output edges. The vectors $in(n)$ and $out(n)$ refer to the incoming and outgoing edges of n in the interprocedural CFG. All statements have one incoming edge and one outgoing edge except for the following cases:

- If $stmtAt(n) = \texttt{merge}$ then $len(in(n)) = 2$ where $in(n)[0]$ and $in(n)[1]$ are the two inputs to the merge. Merge nodes with more than two inputs are split into a sequence of two-input merge nodes.

- If $stmtAt(n) = (\texttt{if}\ b\ \texttt{goto}\ L_1\ \texttt{else}\ L_2)$ then $len(out(n)) = 2$ where $out(n)[0]$ is the true successor and $out(n)[1]$ is the false successor.

- If $stmtAt(n) = (x := f(b))$ then $len(out(n)) = 2$, where $out(n)[0]$ is the intraprocedural edge from the call site n to the return site (the immediate successor of n in the intraprocedural CFG), and $out(n)[1]$ is the interprocedural edge from the call site n to the entry node of f. The entry node of a function is a merge node. Once the interprocedural CFG is build, the entry node of a function may have more than two inputs, and may need to be split into multiple two-input merge nodes.

- If $stmtAt(n) = (\texttt{return}\ x)$ then $len(out(n)) = k$ where k is the number of call sites to the function that n belongs to. I define $callSiteIndex : Node \rightharpoonup \mathbb{N}$ so that, given a return node n, and a call site n' to the procedure containing n, then $out(n)[callSiteIndex(n')]$ is the edge from n to the return site corresponding to n'.

Definition 2 *We use $i, \eta \xrightarrow{n} j, \eta'$ to say that program state η coming along the i^{th} input edge of n steps to η' on the j^{th} output edge of n. The input and output edges are in the interprocedural CFG. The state transition function $\cdot,\cdot \dot{\rightarrow} \cdot,\cdot \subseteq \mathbb{N} \times State \times Node \times \mathbb{N} \times State$ is defined by:*

- *If $stmtAt(n) = (\texttt{decl}\ x)$ then $0, (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle) \xrightarrow{n} 0, (\rho[x \mapsto l], \sigma[l \mapsto uninit], \xi, \langle l_0, l_1, \ldots\rangle)$*
  where $l \notin dom(\sigma)$

- *If $stmtAt(n) = (\texttt{decl}\ x[i])$ then $0, (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots, \rangle) \xrightarrow{n} 0, (\rho[x \mapsto l], \sigma[l \mapsto newInitArray(s), s \mapsto uninit], \xi, \langle l_{\eta(i)}, l_{\eta(i)+1}, \ldots\rangle)$*
  where $\eta = (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots, \rangle)$, $l \notin dom(\sigma)$, $\eta(i) \in \mathbb{N}$, $\eta(i) > 0$, $s = \langle l_0, \ldots, l_{\eta(i)-1}\rangle$, $s \notin dom(\sigma)$

- *If* $stmtAt(n) =$ `skip` *then* $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$

- *If* $stmtAt(n) =$ `merge` *then* $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$ *and* $1, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$

- *If* $stmtAt(n) = (x := e)$ *then* $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma[\eta(\&x) \mapsto \eta(e)], \xi, \mathcal{M})$
  where $\eta = (\rho, \sigma, \xi, \mathcal{M})$, $\eta(x) \in Location \cup Const$ [1], $\eta(e) \in Location \cup Const$ [2]

- *If* $stmtAt(n) = (*x := e)$ *then* $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma[\eta(x) \mapsto \eta(e)], \xi, \mathcal{M})$
  where $\eta = (\rho, \sigma, \xi, \mathcal{M})$, $\eta(x) \in Location$, $\eta(*x) \in Location \cup Const$, $\eta(e) \in Location \cup Const$

- *If* $stmtAt(n) = (x :=$ `new`$)$ *then* $0, (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle) \xrightarrow{n} 0, (\rho, \sigma[\eta(\&x) \mapsto l, l \mapsto uninit], \xi, \langle l_0, l_1, \ldots\rangle)$
  where $\eta = (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle)$, $\eta(x) \in Location \cup Const$, $l \notin dom(\sigma)$

- *If* $stmtAt(n) = (x :=$ `new array[i]`$)$ *then* $0, (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle) \xrightarrow{n}$
  $0, (\rho, \sigma[\eta(\&x) \mapsto l, l \mapsto newInitArray(s), s \mapsto uninit], \xi, \langle l_{\eta(i)}, l_{\eta(i)+1}, \ldots\rangle)$
  where $\eta = (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle)$, $l \notin dom(\sigma)$, $\eta(x) \in Location \cup Const$, $\eta(i) \in \mathbb{N}$, $\eta(i) > 0$, $s = \langle l_0, \ldots, l_{\eta(i)-1}\rangle$, $s \notin dom(\sigma)$

- *If* $stmtAt(n) = (x := p(b))$ *then* $0, (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle) \xrightarrow{n} 1, (\{(y, l)\}, \sigma[l \mapsto \eta(b)], push(f, \xi), \langle l_0, l_1, \ldots\rangle)$
  where $\eta = (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle)$, $y = formal(callee(n))$, $l \notin dom(\sigma)$, $f = (n, \rho(x), \rho)$, $\eta(x) \in Location \cup Const$, $\eta(b) \in Location \cup Const$

- *If* $stmtAt(n) =$ `(if` b `goto` L_1 `else` L_2`)` *then* $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$
  where $(\rho, \sigma, \xi, \mathcal{M})(b) = 1$

- *If* $stmtAt(n) =$ `(if` b `goto` L_1 `else` L_2`)` *then* $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 1, (\rho, \sigma, \xi, \mathcal{M})$
  where $(\rho, \sigma, \xi, \mathcal{M})(b) = 0$

- *If* $stmtAt(n) =$ `(return` x`)` *then* $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} j, (\rho_0, \sigma_0, \xi_0, \mathcal{M})$
  where $pop(\xi) = ((n_0, l_0, \rho_0), \xi_0)$, $j = callSiteIndex(n_0)$, $dom(\rho) = \{x_1, \ldots, x_i\}$, $\sigma_0 = (\sigma / \{\rho(x_1), \ldots, \rho(x_i)\})[l_0 \mapsto (\rho, \sigma, \xi, \mathcal{M})(x)]$, $(\rho, \sigma, \xi, \mathcal{M})(x) \in Location \cup Const$

---

[1] This check, and the ones like it for other statement types, prevent assignments to array values.

[2] This check, and the ones like it for other statement types, prevent array values from being copied.

The $\cdot,\cdot \stackrel{\cdot}{\rightarrow} \cdot,\cdot$ relation defined above is a function: given $i \in \mathbb{N}$, $\eta \in \mathit{State}$, $n \in \mathit{Node}$ and $j \in \mathbb{N}$, there is at most one η' such that $i, \eta \stackrel{n}{\rightarrow} j, \eta'$. The semantics of the Rhodium IL is therefore deterministic. Furthermore, because functions such as $\eta(e)$, *push* and *pop* are partial, and because of the side conditions in the above definition, the $\cdot,\cdot \stackrel{\cdot}{\rightarrow} \cdot,\cdot$ function is partial. For example, if x has not been declared, meaning that $x \notin \mathit{dom}(\rho)$, then $0, (\rho, \sigma, \xi, \mathcal{M})$ cannot step through the statement $x := e$. As another example, call statements cannot step from a state where the remaining stack size is 0, because the *push* operation would not succeed.

A *machine configuration* is a pair $\delta = (e, \eta)$ where $e \in \mathit{Edge}$ and $\eta \in \mathit{State}$. Here e indicates where control has reached, and η represents the program state. I denote by *MachineConfig* the set of all machine configurations.

Definition 3 *Given a program π, the machine configuration transition function $\rightarrow_\pi \subseteq \mathit{MachineConfig} \times \mathit{MachineConfig}$ is defined by:*

$$(e, \eta) \rightarrow_\pi (e', \eta') \Leftrightarrow h, \eta \stackrel{n}{\rightarrow} h', \eta'$$
where $n = \mathit{dst}_{CFG(\pi)}(e)$, $h = \mathit{inIndex}_{CFG(\pi)}(e, n)$, $h' = \mathit{outIndex}_{CFG(\pi)}(e', n)$

The $\rightarrow_\pi^*$ relation is the reflexive, transitive closure of the $\rightarrow_\pi$ relation.

The intraprocedural state transition function $\hookrightarrow$ is similar to $\rightarrow$, except that it operates over the intraprocedural CFG.

Definition 4 *The intraprocedural state transition function $\cdot,\cdot \stackrel{\cdot}{\hookrightarrow} \cdot,\cdot \subseteq \mathbb{N} \times \mathit{State} \times \mathit{Node} \times \mathbb{N} \times \mathit{State}$ is defined by:*

- *If $\mathit{stmtAt}(n)$ is neither a procedure call nor a return, then $i, \eta \stackrel{n}{\hookrightarrow} j, \eta'$*
  where $i, \eta \stackrel{n}{\rightarrow} j, \eta'$

- *If $\mathit{stmtAt}(n) = (x := f(b))$ then $0, \eta \stackrel{n}{\hookrightarrow} 0, \eta'$*
  where $\pi = \mathit{prog}(n)$, $0, \eta \stackrel{n}{\rightarrow} 1, \eta_p$, and $(\mathit{out}(n)[1], \eta_p) \rightarrow_\pi \ldots \rightarrow_\pi (e, \eta')$ is the shortest trace such that $\mathit{stack}(\eta') = \mathit{stack}(\eta)$

- *If $\mathit{stmtAt}(n) = (\texttt{return}\ x)$ then $0, \eta \stackrel{n}{\hookrightarrow} 0, \eta'$*
  where $0, \eta \stackrel{n}{\rightarrow} j, \eta'$ for some j (by the definition of $\rightarrow$, there can be at most one such j if η is given).

The only part of the semantics that will be used in the rest of this dissertation is the intraprocedural state transition function $\hookrightarrow$ from Definition 4. In particular, the soundness conditions for propagation and transformation rules will be defined in terms of $\hookrightarrow$. Furthermore, all the theory that is done by hand once will treat the $\hookrightarrow$ function as a black box. The details of $\hookrightarrow$, and more generally the details of the Rhodium IR semantics, have an effect solely on the background axioms that the theorem prover uses to discharge soundness obligations.

Chapter 4

ANALYSIS FRAMEWORK

Chapter 3 presented the IR over which Rhodium optimizations run. This chapter presents a generic framework for defining analyses and transformations over this IR. This framework is used in Chapters 5 and 6 to define the semantics of Rhodium optimizations. The analysis and transformation framework presented here is based on abstract interpretation [29], and it is adapted from a previous framework for composing dataflow analyses and transformations [63]. The part of the framework described in this chapter is the core definition of analyses and transformations, on top of which the composition functionality is built. By using the composing framework to formalize Rhodium optimizations, the theorems from [63] can be adapted to show that Rhodium analyses and transformations can be composed soundly, while allowing them to interact in mutually beneficial ways.

4.1 Definition

An *analysis* is a tuple $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ where $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ is a complete lattice, $\alpha : D_c \to D$ is the abstraction function, and $F : Node \times D^* \to D^*$ is the flow function for nodes. The elements of D, the domain of the analysis, are dataflow facts about edges in the IR (which correspond to program points in our CFG representation). The flow function F provides the interpretation of nodes: given a node and a tuple of input dataflow values, one per incoming edge to the node, F produces a tuple of output dataflow values, one per outgoing edge from the node. D_c is the domain of a distinguished analysis, the concrete analysis $\mathcal{C} = (D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c, id, F_c)$, which specifies the concrete semantics of the program. The flow function F_c, which specifies the concrete behavior of IR nodes, must be monotonic, and it must also be bottom-preserving, meaning that $F_c(\bot_c) = \bot_c$. The concrete analysis is left unspecified by the framework, leaving the choice up to the client. Various options are available for defining this concrete semantics, including forward

and backward collecting state semantics, and forward and backward collecting trace semantics. The formalization of forward Rhodium optimizations, given in Chapter 5, uses a forward collecting state semantics, and the formalization of backward optimizations, given in Chapter 6, uses a backward collecting state semantics.

Furthermore, the framework requires each abstraction function α to be join-monotonic, which means that for any chain Y of elements (a sequence of ascending elements) in D_c, it is the case that $\alpha(\bigsqcup_c Y) \sqsubseteq \bigsqcup\{\alpha(d) \mid d \in Y\}$. This property is weaker than the standard requirement in abstract interpretation that α be continuous, which says that for any chain Y of elements in D_c, it is the case that $\alpha(\bigsqcup_c Y) = \bigsqcup\{\alpha(d) \mid d \in Y\}$. The abstraction function α must also be bottom-preserving, meaning that $\alpha(\bot_c) = \bot$.

The solution of an analysis $\mathcal{A}$ over a domain D is provided by the function $S_{\mathcal{A}} : \mathit{Graph} \times D^* \to (\mathit{Edge} \to D)$. Given a graph g and a tuple of abstract values for the input edges of g, $S_{\mathcal{A}}$ returns the final abstract value for each edge in g. This is done by initializing all edges in g to $\bot$ (except for input edges, which are initialized to the given input values), and then applying the flow functions of $\mathcal{A}$ until a fixed point is reached. A detailed definition of $S_{\mathcal{A}}$ is in Appendix B.[1]

The framework described here is intraprocedural in nature, and as a result the formalism and proofs are simpler if the concrete semantics of a call node skips over the call, rather than steps into it. To this end, the framework provides a definition of the concrete semantics of calls that does exactly this. In particular, $S_{\mathcal{C}}$ is used recursively to solve the callee's intraprocedural CFG, and then the result on the outgoing edge is propagated to the caller.

Definition 5 *The concrete flow function for a node n for which $\mathit{stmtAt}(n) = (x := f(b))$ is defined by the framework as:*

$$F_c(n, cs)[0] = \mathit{calleeToCaller}(n, \overrightarrow{S_{\mathcal{C}}(\mathit{cfg}(\mathit{callee}(n)), \iota)}(\mathit{OutEdges}_{\mathit{cfg}(\mathit{callee}(n))}))$$
$$\text{where } \iota = \mathit{callerToCallee}(n, cs)$$

In the above definition, the function $\mathit{callerToCallee} : \mathit{Node} \times D_c^* \to D_c^*$ translates the concrete information from the caller's call site to the callee's entry point, and the function

[1] Although the concrete solution function $S_{\mathcal{C}}$ is usually not computable, the mathematical definition of $S_{\mathcal{C}}$ is still perfectly valid. Our framework does not evaluate $S_{\mathcal{C}}$; I use $S_{\mathcal{C}}$ only to formalize the soundness of analyses.

callerToCallee : $Node \times D_c^* \rightarrow D_c^*$ transforms the concrete information from the caller's return statement to the callee's return site. These functions must be defined by the client of the framework, along with the concrete analysis $\mathcal{C}$. The framework requires *callerToCallee* and *calleeToCaller* to have two properties: (1) they cannot depend on any property of the node at which they are evaluated, except for the statement at that node, and (2) they must be monotonic in their second argument. Formally, this is stated in the following four conditions:

$$\begin{aligned}&\forall(n, n', cs) \in Node \times Node \times D_c^* \ . \\ &\quad stmtAt(n) = stmtAt(n') \Rightarrow callerToCallee(n, cs) = callerToCallee(n', cs)\end{aligned} \tag{4.1}$$

$$\begin{aligned}&\forall(n, n', cs) \in Node \times Node \times D_c^* \ . \\ &\quad stmtAt(n) = stmtAt(n') \Rightarrow calleeToCaller(n, cs) = calleeToCaller(n', cs)\end{aligned} \tag{4.2}$$

$$\begin{aligned}&\forall(n, cs_1, cs_2) \in Node \times D_c^* \times D_c^* \ . \\ &\quad cs_1 \sqsubseteq_c cs_2 \Rightarrow callerToCallee(n, cs_1) \sqsubseteq_c callerToCallee(n, cs_2)\end{aligned} \tag{4.3}$$

$$\begin{aligned}&\forall(n, cs_1, cs_2) \in Node \times D_c^* \times D_c^* \ . \\ &\quad cs_1 \sqsubseteq_c cs_2 \Rightarrow calleeToCaller(n, cs_1) \sqsubseteq_c calleeToCaller(n, cs_2)\end{aligned} \tag{4.4}$$

An *Analysis followed by Transformations*, or an *AT-analysis* for short, is a pair $(\mathcal{A}, R)$ where $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ is an analysis, and $R : Node \times D^* \rightarrow Graph \cup \{\epsilon\}$ is a *local replacement function*. The local replacement function R specifies how a node should be transformed after the analysis has been solved. Given a node n and a tuple of elements of D representing the final dataflow analysis solution for the input edges of n, R either returns a graph with which to replace n, or ϵ to indicate that no transformation should be applied to this node. To be syntactically valid, a replacement graph must have the same number of input and output edges as the node it replaces, and its nodes and edges must be unique (so that splicing a replacement graph into the enclosing graph does not cause conflicts). I denote by RF_D the set of all replacement functions over the domain D, or in other words $RF_D = Node \times D^* \rightarrow Graph \cup \{\epsilon\}$.

After analysis completes, the intermediate representation is transformed in a separate pass by a transformation function $T : RF_D \times Graph \times (Edge \rightarrow D) \rightarrow Graph$. Given a replacement function R, a graph g, and the final dataflow analysis solution, T replaces each

node in g with the graph returned by R for that node, thus producing a new graph. Since nodes cannot be shared across programs (because otherwise the *prog* function would be ill-defined), the graph returned by T cannot contain nodes from the input graph g. To this end, nodes in replacement graphs are required to be fresh, and the T function must create a fresh copy of nodes for which R returned ϵ. A detailed definition of T can be found in Appendix B.

Finally, the effect of an AT-analysis $(\mathcal{A}, R)$ on a program is given by the function $[\![\mathcal{A}, R]\!] : Prog \to Prog$ defined as follows, where the last condition defines the *prog* field for nodes in the transformed program π':

Definition 6 *If* $\pi = (p_1, \ldots, p_n)$ *then* $[\![\mathcal{A}, R]\!](\pi) = \pi'$ *where:*

$$\begin{array}{lcll} \pi' & = & (p'_1, \ldots, p'_n) & \\ p'_i & = & proc(name(p_i), formal(p_i), r_i) & \text{for } i \in [1..n] \\ r_i & = & T(R, g_i, S_{\mathcal{A}}(g_i, \top)) & \text{for } i \in [1..n] \\ g_i & = & cfg(p_i) & \text{for } i \in [1..n] \\ \top & = & \text{top element of the lattice of } \mathcal{A} & \\ \forall n \in N_{r_i} \;.\; prog(n) = \pi' & & & \text{for } i \in [1..n] \end{array}$$

4.2 Soundness

An AT-analysis $(\mathcal{A}, R)$ is sound if each CFG produced by $(\mathcal{A}, R)$ has the same concrete semantics as the corresponding original CFG. This is formalized in the following definition of soundness of $(\mathcal{A}, R)$:

Definition 7 *Let* $(\mathcal{A}, R)$ *be an AT-analysis, let* $\pi = (p_1, \ldots, p_n)$ *be a program where* $g_i = cfg(p_i)$*, and let* $[\![\mathcal{A}, R]\!](\pi) = \pi'$ *where* $\pi' = (p'_1, \ldots, p'_n)$ *and* $r_i = cfg(p'_i)$*. We say that* $(\mathcal{A}, R)$ *is sound iff:*

$$\forall i \in [1..n] \;.\; \forall \iota_c \in D_c \;.\; \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(OutEdges_{r_i})$$

The above definition of soundness does not require the concrete semantics to be exactly preserved. Rather, the original and the transformed concrete semantics must be related by $\sqsubseteq_c$. In general, it is well understood that proving exact preservation of semantics is not always practical. Indeed, compilers may want to transform code in ways that preserve the behavior of correct programs, but not necessarily incorrect programs. As a simple example,

some compilers will remove an assignment $x := *y$ if x is never used afterward, even though this changes the behavior of the program if y happens to be null. The standard way of addressing this issue is to use a refinement relation $\sqsubseteq_{ref}$ to determine when an original program is correctly "implemented" by a transformed program.

For the definition of soundness of an AT-analysis, I have chosen this refinement relation to be $\sqsubseteq_c$, and as a result, the optimizer is required only to preserve the behavior of runs that don't get stuck. For example, if the original program contained a statement $x := *y$ that would get stuck on some runs (because of y being null), the optimizer would be allowed to remove this statement, thus producing a program that gets stuck less often. On the other hand, the $\sqsubseteq_c$ refinement relation requires executions that don't get stuck to be preserved, and so the optimizer would not be allowed to transform a program in a way that makes it stuck more often. Another way of understanding this stems from the fact that, when a statement is stuck in a program state, its semantics is unspecified in that program state. As a result, the $\sqsubseteq_c$ refinement relation is simply stating that optimizations must preserve the semantics of runs whose semantics is specified.

Similarly to C, the semantics of the current Rhodium IL is underspecified, meaning that some executions get stuck. However, the degree to which the semantics of the Rhodium IL is specified (or unspecified) only affects the definition of the IL semantics, namely the $\hookrightarrow$ relation from Section 3.4. Since the IL semantics is treated as a black box, the Rhodium system can easily be adapted to model programming languages whose semantics are more or less specified. For example, to model a programming language like Java where a null pointer dereference creates an exception that must be preserved by the optimizer, one only needs to change the semantics of the IL so that a null pointer dereference produces a special error value instead of getting stuck. The semantics of dereferencing a null pointer would then be specified, as opposed to unspecified, and would therefore have to be preserved by the optimizer.

I define here two conditions that together are sufficient to show that an AT-analysis $(\mathcal{A}, R)$ is sound. First, the flow function of $\mathcal{A}$ must be sound according to the following definition:

Definition 8 *Given an analysis $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$, we say that its flow function*

F_a is sound iff it satisfies the following property:

$$\begin{array}{l}\forall(n, cs, ds) \in Node \times D_c^* \times D_a^* \ . \\ \quad \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_a(n, ds)\end{array} \tag{4.5}$$

Informally, this states that if some abstract fact ds approximates some concrete fact cs, then the result of applying F to ds should approximate the result of applying F_c to cs. If the flow function of an analysis $\mathcal{A}$ is sound, then it is possible to show that $\mathcal{A}$ is sound, meaning that its solution conservatively approximates the solution of the concrete analysis $\mathcal{C}$. This is formalized by the following definition and theorem, the latter of which is proved in Appendix B.

Definition 9 *We say that an analysis $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$ is sound iff:*

$$\begin{array}{l}\forall(g, \iota_c, \iota_a) \in Graph \times D_c^* \times D_a^*. \\ \quad \overrightarrow{\alpha}(\iota_c) \sqsubseteq \iota_a \Rightarrow \widetilde{\alpha}(S_{\mathcal{C}}(g, \iota_c)) \sqsubseteq S_{\mathcal{A}}(g, \iota_a)\end{array}$$

Theorem 4 *If the flow function of an analysis $\mathcal{A}$ is sound then $\mathcal{A}$ is sound.*

Property (4.5) is sufficient for proving Theorem 4. Moreover it is weaker than the local consistency property of Cousot and Cousot (property 6.5 in [29]), which is:

$$\begin{array}{l}\forall(n, cs, ds) \in Node \times D_c^* \times D_a^*. \\ \quad \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_a(n, \overrightarrow{\alpha}(cs))\end{array}$$

Indeed, the above property and the monotonicity of F_a imply property (4.5). I use the weaker condition (4.5) because in this way the formalization of soundness does not depend on the monotonicity of F_a. Although the flow function induced by a set of Rhodium rules is guaranteed to be monotonic (as discussed in Section 2.1.3), when the framework presented here is used for composing program analyses and transformations (as shown in [63]), the flow function F_a is in fact generated by the framework, and reasoning about its monotonicity requires additional effort on the part of the analysis writer (see Sections 5 and 6 of [63]). By decoupling the soundness result from the monotonicity of F_a, it is therefore possible to guarantee soundness even if F_a has not been shown to be monotonic.[2]

[2]Termination in the face of a non-monotonic flow function generated by the composing framework is discussed in Section 4.3 of [63].

In addition to the analysis's flow function having to be sound, R must produce graph replacements that are semantics-preserving. This is formalized by requiring that the replacement function R be sound according to the following definition:

Definition 10 *We say that a replacement function R in $(\mathcal{A}, R)$ is sound iff it satisfies the following property, where $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$:*

$$\begin{array}{l} \forall (n, ds, g) \in Node \times D_a^* \times Graph. \\ \quad R(n, ds) = g \Rightarrow \\ \quad\quad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ \quad\quad\quad F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)] \end{array} \tag{4.6}$$

Property (4.6) requires that if R decides to replace a node n with a graph g on the basis of some analysis result ds, then for all possible input tuples of concrete values consistent with ds, if evaluation of n on these inputs succeeds, then it must be the case that n and g compute exactly the same output tuple of concrete values. It is not required that n and g produce the same output for all possible inputs, just those consistent with ds. For example, if $\mathcal{A}$ determines that all stores coming into a node n will always assign some variable `x` a value between 1 and 100 then n and g are not required to produce the same output for any store in which `x` is assigned a value outside of this range.

If both the flow function and the replacement function of an AT-analysis are sound, then it is possible to show that the AT-analysis is sound, meaning that the final graph it produces has the same concrete behavior as the original graph, whenever evaluation of the original graph succeeds. This is stated in the following theorem, which is proved in Appendix B:

Theorem 5 *Given an AT-analysis $(\mathcal{A}, R)$, where $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$, if F_a and R are sound, then $(\mathcal{A}, R)$ is sound.*

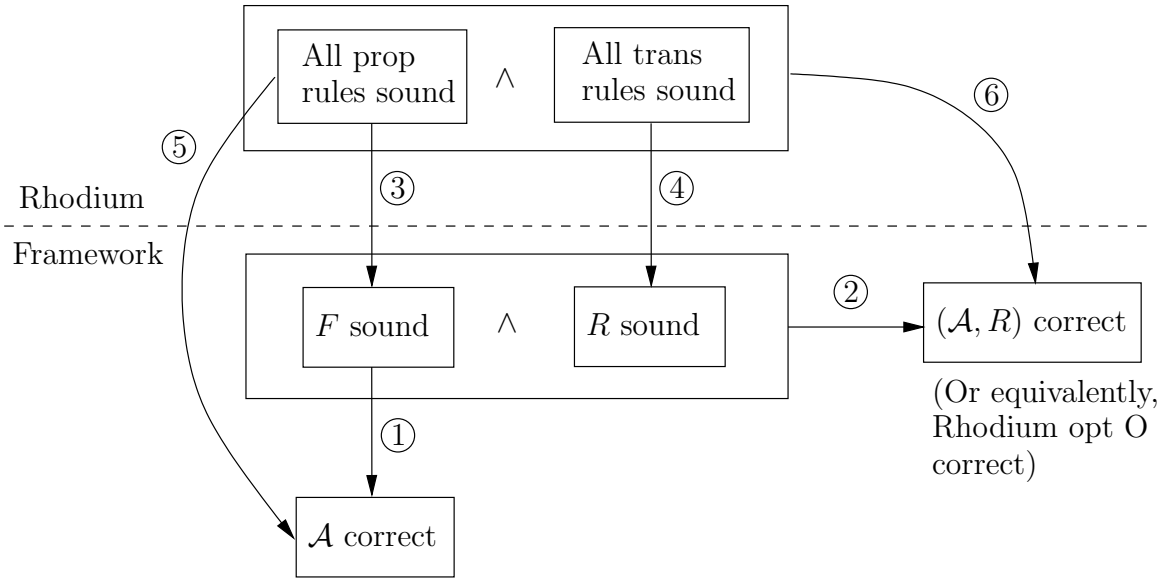
One important property of the analysis framework presented here is that, if the replacement function R is sound, then the selected transformations do not interfere with each other – it is sound to perform all of them in parallel. This allows multiple transformations to be performed without having to reanalyze the program after each transformation. The framework has an even stronger property, called subset-soundness, which is that any subset of the selected transformations can be performed soundly. Intuitively, subset-soundness follows from the fact that the soundness condition (4.6) requires R to preserve semantics

only when it returns a non-ϵ value. Therefore, returning ϵ more often (which corresponds to selecting a subset of the transformations) preserves the soundness of R. In Rhodium's predecessor, Cobalt [64], subset-soundness was essential to making Cobalt's profitability heuristics work (see [64] for more details on Cobalt's profitability heuristics). Rhodium's way of expressing profitability heuristics, however, *does not* require the subset-soundness property, even though the framework provides it nonetheless.

The diagram in Figure 4.1 displays the connections among the various lemmas and theorems of this dissertation. Each rectangular box in the diagram represents a predicate, with inner boxes representing sub-predicates. For example, the large box in the middle of the diagram represents the predicate "F is sound and R is sound". Arrows between boxes represent logical implication, and labels on arrows, combined with the legend at the bottom of the figure, indicate what lemma or theorem states the implication.

The diagram in Figure 4.1 is split into two parts, separated by a dashed line: the bottom part refers to predicates and theorems stating properties of the generic analysis framework presented in this section; the top part, which will be covered in more detail in Chapters 5 and 6, refers to predicates and theorems specific to the Rhodium system.

Some lemmas and theorems in the legend are labeled "Forward", "Backward" or "Informal", to distinguish between various versions of the same lemma or theorem. "Informal" refers to the informal lemmas and theorems used in the overview of the Rhodium soundness checker from Section 2.2. "Forward" refers to a forward version of the lemma or theorem, which applies only to forward optimizations, and "Backward" refers to a backward version, which applies only to backward optimizations.



① Informal: Theorem 1 (Section 2.2.1)
Theorem 4 (Section 4.2)

② Theorem 5 (Section 4.2)

③ Informal: Lemma 1 (Section 2.2.1)
Forward: Lemma 3 (Section 5.4.3)
Backward: Lemma 6 (Section 6.3.3)

④ Forward: Lemma 4 (Section 5.4.3)
Backward: Lemma 7 (Section 6.3.3)

⑤ Informal: Theorem 2 (Section 2.2.1)
Forward: Theorem 8 (Section 5.4.3)
Backward: Theorem 10 (Section 6.3.3)

⑥ Informal: Theorem 3 (Section 2.2.2)
Forward: Theorem 7 (Section 5.4.3)
Backward: Theorem 9 (Section 6.3.3)

Figure 4.1: Connections among the various lemmas and theorems of this dissertation

Chapter 5

FORWARD RHODIUM OPTIMIZATIONS

This chapter presents the formal details of how forward Rhodium optimizations are checked for soundness. To define soundness, one must first define the concrete semantics that optimizations must preserve. This concrete semantics is presented in Section 5.2. One must also have a precise definition of how Rhodium optimizations run. To this end, the formal semantics of forward Rhodium optimizations is given in Section 5.3. Finally, Section 5.4 describes the details of how the soundness checker works. In particular, it presents formal versions of the local soundness conditions (prop-sound) and (trans-sound), and also shows how these conditions are formalized inside the theorem prover. Before going into all these details, the chapter begins in Section 5.1 with a formal definition of the Rhodium syntax, which will make later parts of the chapter easier to understand.

5.1 Rhodium Syntax

The grammar of the full Rhodium language (not just the forward subset) is given in Figures 5.1 and 5.2. I use $E*$ to represent zero or more repetitions of E, and E/X to represent zero or more repetitions of E separated by X. Non-terminals are in written in *italics* font, and terminals are written either in `typewriter` font or **boldface** font. The non-terminals *Stmt* and *Expr* refer to statements and expression of the *extended intermediate language*, which augments the intermediate language from Figure 3.1 with Rhodium variables. In particular, each production in the grammar of the original intermediate language is extended with a case for a Rhodium variable, as shown in Figure 5.3. The extended intermediate language therefore includes such statements as $X := A\ OP\ B$, where X, A, OP and B are Rhodium variables ranging over pieces of syntax of the intermediate language. The extended intermediate language also contains the special `merge` statement, so that programmers can write user-defined merge operations. Finally, the extended intermediate language does not

RhodiumProg	::=	*Decl*∗
Decl	::=	*VarDecls* | *EdgeFactDecl* | *VirtualEdgeFactDecl*
	|	*NodeFactDecl* | *PropagateRule* | *TransformRule*
VarDecls	::=	**decl** *VarDecl*/, **in** *Decl* ∗ **end**
EdgeFactDecl	::=	**define forward edge fact** *id* (*VarDecl*/,) **with meaning** *Meaning*
	|	**define backward edge fact** *id* (*VarDecl*/,) **with meaning** *Meaning*
VirtualEdgeFactDecl	::=	**define virtual edge fact** *id* (*VarDecl*/,) **@***RhodiumVar* = ψ
NodeFactDecl	::=	**define node fact** *id* (*VarDecl*/,) = ψ
VarDecl	::=	*RhodiumVar* : τ
Meaning	::=	`true` | `false` | `!` *Meaning* | *Meaning* `||` *Meaning*
	|	*Meaning* `&&` *Meaning* | *Meaning* `=>` *Meaning*
	|	`forall` *VarDecl*/, . *Meaning* | `exists` *VarDecl*/, . *Meaning*
	|	*MeaningTerm* `==` *MeaningTerm*
	|	*MeaningTerm* `!=` *MeaningTerm*
	|	*MeaningPredSymbol* (*MeaningTerm*/,)
MeaningTerm	::=	*Expr* | *Stmt* | `eta` | `eta_1` | `eta_2`
	|	*MeaningFunSymbol* (*MeaningTerm*/,)
MeaningPredSymbol	::=	`isStmtNotStuck` | `isExprNotStuck` | `isLoc`
	|	`isConst` | `equalUpTo`
MeaningFunSymbol	::=	`evalExpr` | `newConst` | `getConst` | `applyUnaryOp`
	|	`applyBinaryOp` | `min` | `max`
PropagateRule	::=	**if** ψ **then** *EdgePred*
TransformRule	::=	**if** ψ **then transform to** *Stmt*

Figure 5.1: Rhodium syntax (continued in Figure 5.2)

ψ	::=	`true` | `false` | `!` ψ | ψ `||` ψ | ψ `&&` ψ | ψ `=>` ψ
	|	`forall` *VarDecl*/, . ψ | `exists` *VarDecl*/, . ψ
	|	*Term* `==` *Term* | *Term* `!=` *Term* | *EdgePred* | *NodePred*
	|	**case** *Term* **of** (*Term* `=>` ψ) $*$ **endcase**
NodePred	::=	*id* (*Term*/,)
EdgePred	::=	*id* (*Term*/,) @*Edge*
Term	::=	*Expr* | *Stmt* | `currStmt` | `currNode`
	|	*PrimFunSymbol* (*Term*/,)
PrimFunSymbol	::=	`newConst` | `getConst` | `applyUnaryOp` | `applyBinaryOp` | `min` | `max`
Edge	::=	`in` | `out` | `in` [*EdgeIndex*] | `out` [*EdgeIndex*]
EdgeIndex	::=	*IntLiteral* | `true` | `false`
RhodiumVar	::=	*id*
id	::=	identifiers
τ	::=	`Stmt` | `Expr` | `CallExpr` | `NonCallExpr` | `LHSExpr` | `BaseExpr`
	|	`Const` | `Var` | `Deref` | `Ref` | `Fun` | `Label` | `UnaryOp` | `BinaryOp`
	|	`Edge` | `HeapSummary` | `CFGNode` | `Loc`

Figure 5.2: Rhodium syntax (continued from Figure 5.1)

Stmt	::=	... (minus label statement form) | *RhodiumVar* | `merge`
Expr	::=	... | *RhodiumVar*
BaseExpr	::=	... | *RhodiumVar*
Constant	::=	... | *RhodiumVar*
ILVar	::=	... | *RhodiumVar*
ProcName	::=	... | *RhodiumVar*
Label	::=	... | *RhodiumVar*
OP	::=	... | *RhodiumVar*

Figure 5.3: Rhodium extended intermediate language, as an extension to the grammar from Figure 3.1

contain label statements, since these statements have already been removed from the CFG.

Precedence for logical operators is as follows (from most binding to least binding): negation (`!`), binary logical operators (`||`, `&&`, `=>`), quantifiers (`exists`, `forall`) and case expressions. Unlike in standard logic, all binary logical operators in Rhodium associate to the left. Thus, a || b && c associates as (a || b) && c. Programmers can insert brackets to override the default associativity. The bracket syntax is not shown in the grammar – I assume that these brackets are just used by the parser to determine which parse tree of the unbracketed grammar to generate.

The type τ of a Rhodium variable indicates what the variable ranges over. There are a variety of types that range over ASTs of the C-like Rhodium intermediate language, including statements (`Stmt`), various expression forms (`Expr`, `CallExpr`, `NonCallExpr`, `LHSExpr`, `BaseExpr`, `Const`, `Var`, `Deref`, `Ref`), function names (`Fun`), labels (`Label`), unary operators (`UnaryOp`), and binary operators (`BinaryOp`). The type `Edge` is used for Rhodium variables ranging over edge names – these variables are used in virtual edge fact declarations. The `HeapSummary` type ranges over heap summaries[1], and the `CFGNode` type ranges over nodes in the CFG (the `currNode` term is of type `CFGNode`). Finally, the `Loc` type ranges over locations in the store. The Rhodium system performs some simple typechecking to make sure that edge facts, node facts, and primitives are used in type-correct ways.

Figures 5.1 and 5.2 present the syntax that will be used in the formal details of Chapters 5 and 6. The syntax implemented in the Rhodium system is slightly different from the one in Figures 5.1 and 5.2 – the real syntax has a few additions for making parsing easier. Furthermore, the snippets of code presented in figures of this dissertation have used (and will continue to use) a few simple aesthetic sugars. In particular, I use math symbols rather than text, for example $\wedge$ instead of `&&`, $\vee$ instead of `||`, $\Rightarrow$ instead of `=>`, and η instead of `eta`; I use mathematical sugars for some primitives, for example $\eta(E)$ instead

[1]The heap summary type is defined by the user with a dynamic semantics extension (see Section 2.4), and so in theory it should not appear in the Rhodium syntax. However, dynamic semantic extensions are currently expressed using the primitive language of Simplify, and as a result the Rhodium typechecker, which makes sure that fact parameters have the right type, is not aware of such user-defined types. As a temporary solution until the parser for semantic extensions is written, I have included the `HeapSummary` type in the syntax of Rhodium.

of `evalExpr`(`eta`, E), and $\eta_1/X = \eta_2/X$ instead of `equalUpTo`(X, `eta_1`, `eta_2`);[2] finally, I sometimes use a different font, for example, *in* and *out* instead of `in` and `out`.

5.2 Concrete semantics

To instantiate the analysis framework from Chapter 4, one must specify the concrete analysis $\mathcal{C}$, which defines the IR semantics that optimizations are meant to preserve. For forward optimizations, I have chosen this concrete semantics to be a forward state collecting semantics. This means that the concrete analysis computes, for each edge in a procedure's CFG, the set of states that the procedure can be in at that edge, given some initial set of states at the beginning of the procedure. The concrete flow function simply steps all incoming program states using the intraprocedural step function $\hookrightarrow$ (from Definition 4). These ideas are made precise in the following definition of the forward concrete analysis:

Definition 11 (forward concrete analysis) *The forward concrete analysis* $\mathcal{C} = (D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c, id, F_c)$ *is defined by:*

$$(D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c) = (2^{State}, \cup, \cap, \subseteq, State, \emptyset)$$

and

$$F_c(n, cs)[h] = \{\eta \mid \exists\ \eta' \in State, i \in \mathbb{N}\ .\ [\eta' \in cs[i]\ \wedge\ i, \eta' \overset{n}{\hookrightarrow} h, \eta]\} \tag{5.1}$$

The choice of a state collecting semantics has implications with respect to the guarantees that the Rhodium system provides. The state collecting semantics computes set of states at each edge, ignoring the order in which they are executed. In theory, this allows for flexibility in reordering executions. For example, on inputs that do not cause the execution to get stuck or to loop forever, the state collecting semantics guarantees that the input-state-to-output-state behavior of the procedure is preserved, even if at internal edges in the procedure, the order in which states are executed has changed. Unfortunately, for inputs that cause the original procedure not to terminate, the state collection semantics may not provide a strong enough guarantee. For example, if a procedure has an infinite loop, a state collecting semantics would allow iterations of the loop to be re-ordered, which may

[2]The interpretation of $\eta_1/X = \eta_2/X$ as a meaning for a backward fact schema will be explained in Section 6.2.3.

or may not be appropriate. In the context of the Rhodium system, one could provide a stronger guarantee for infinite runs by adopting a *trace* collecting semantics. The soundness conditions for propagation and transformation rules would remain the same, but the proofs that are done by hand would have to be updated to take the new concrete semantics into account.

Along with the concrete analysis, one must also specify the *callerToCallee* and *calleeToCaller* functions, which are used by the framework to define the concrete flow function for call nodes. For the forward case, these two functions are defined as follows:[3]

Definition 12 (forward *callerToCallee***)** *The forward version of callerToCallee is defined by:*

$$\begin{aligned} &callerToCallee(n, cs)[0] = \{\eta \mid \exists\ \eta' \in State\ .\ [\eta' \in cs[0]\ \wedge\ 0, \eta' \xrightarrow{cn} 1, \eta]\} \\ &\text{where } cn = interCallNode(n) \end{aligned}$$

Definition 13 (forward *calleeToCaller***)** *The forward version of calleeToCaller is defined by:*

$$calleeToCaller(n, cs) = cs$$

It is easy to see from the above definitions that *calleeToCaller* and *callerToCallee* satisfy requirements (4.1), (4.2), (4.3), and (4.4).

As shown in Definition 5 of Section 4.1, the framework has already defined the concrete flow function for call nodes as follows:

$$\begin{aligned} &F_c(n, cs)[0] = calleeToCaller(n, \overrightarrow{S_{\mathcal{C}}(cfg(callee(n)), \iota)}(OutEdges_{cfg(callee(n))})) \\ &\text{where } \iota = callerToCallee(n, cs) \end{aligned} \tag{5.2}$$

The concrete forward flow function defined in (5.1) computes the same information as the one in (5.2) at call nodes. Indeed, Equation (5.1) says that the set of states after a call node is computed by taking the union over all incoming states of the resulting outgoing state. Equation (5.2) describes the same computation, except that it uses the collecting semantics of the callee to determine the set of all outgoing states.

---

[3]Recall that *callerToCallee* and *calleeToCaller* get evaluated only on call nodes in the intraprocedural CFG, and that *interCallNode* returns the interprocedural call node corresponding to an intraprocedural call node (see Section 3.3 and Figures 3.4 and 3.3)

5.3 Rhodium semantics

This section presents the semantics of a forward Rhodium optimization O, which is an optimization containing only forward propagation and transformation rules. The semantics of O is defined using an associated AT-analysis $(\mathcal{A}_O, R_O)$, where $\mathcal{A}_O = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, F_O, \alpha)$. I first define the lattice $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ over which the analysis runs, followed by the flow function F_O, the abstraction function α, and finally the replacement function R_O.

In the remainder of this chapter, I fix the forward Rhodium optimization O whose semantics is being defined. I also assume that all node facts and virtual edge facts in O have been macro-expanded away. In particular, each occurrence of a node fact or virtual edge fact is replaced by an instance of its defining body, with formals replaced by actual parameters. Furthermore, I use $\mathit{EdgeFact}_O$ to denote the set of edge-fact schema names declared in O, I use $\mathit{arity}(\mathit{EF})$ to denote the number of arguments to a fact schema $\mathit{EF} \in \mathit{EdgeFact}_O$, and I use $r \in O$ to say that rule r appears in O. For example, (**if** ψ **then** ϕ) $\in O$ says that O contains the propagation rule **if** ψ **then** ϕ. Finally, I assume that the edge names *in* and *out* have been desugared in the manner described in Appendix A.

5.3.1 Lattice

The forward Rhodium AT-analysis runs over the power-set lattice of all Rhodium forward facts. An element of this lattice is a set of facts, and elements are ordered using the subset relation. The most precise element ($\bot$) is the full set of facts, and the most conservative element ($\top$) is the empty set. This lattice is formalized in the following definition, where *GroundTerm* is the set of all terms that do not contain Rhodium variables, primitives, `currStmt`, or `currNode`:

Definition 14 (lattice for forward AT-analysis) *The lattice of* $(\mathcal{A}_O, R_O)$ *is:*

$$(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot) = (2^{\mathit{Facts}}, \cap, \cup, \supseteq, \emptyset, \mathit{Facts})$$

where

$$\mathit{Facts} = \{\mathit{EF}(t_1, \ldots, t_i) \mid \mathit{EF} \in \mathit{EdgeFact}_O \wedge i = \mathit{arity}(\mathit{EF}) \wedge (t_1, \ldots, t_i) \in \mathit{GroundTerm}^i\}$$

5.3.2 Flow function

The flow function $F_O : \mathit{Node} \times D^* \to D^*$ uses the propagation rules from O to determine what outgoing facts to return. In particular, $F_O(n, ds)$ finds all propagation rules that trigger at node n given incoming facts ds, and returns the facts propagated by those rules:

Definition 15 (forward flow function) *The flow function F_O of $(\mathcal{A}_O, R_O)$ is defined by:*

$$F_O(n, ds)[h] = \{\theta(EF(t_1, \ldots, t_i)) \mid \begin{array}{l}(\textbf{if } \psi \textbf{ then } EF(t_1, \ldots, t_i)@\texttt{out}[h]) \in O \wedge \\ [\![\psi]\!](\theta, ds, n)\}\end{array} \tag{5.3}$$

In the above definition, θ is a substitution mapping Rhodium variables to ground terms. I use *Subst* to represent the set of all substitutions, $\theta(\cdot)$ to represent substitution application, and $\theta[x \mapsto y]$ to represent the substitution identical to θ, but with x mapping to y.

The meaning of the antecedent ψ of a forward rule is given by a function $[\![\psi]\!] : \mathit{Subst} \times D^* \times \mathit{Node} \to \mathit{bool}$. Intuitively, $[\![\psi]\!](\theta, ds, n)$ is true iff $\theta(\psi)$ holds at node n, given incoming facts ds. The following definition makes this notion precise:

Definition 16 *The meaning function $[\![\psi]\!] : \mathit{Subst} \times D^* \times \mathit{Node} \to \mathit{bool}$ for the antecedent ψ of a forward rule is defined as follows:*

$$\begin{array}{lcl}
[\![\texttt{true}]\!](\theta, ds, n) & = & \mathit{true} \\
[\![\texttt{false}]\!](\theta, ds, n) & = & \mathit{false} \\
[\![!\ \psi]\!](\theta, ds, n) & = & \neg[\![\psi]\!](\theta, ds, n) \\
[\![\psi_1\ ||\ \psi_2]\!](\theta, ds, n) & = & [\![\psi_1]\!](\theta, ds, n) \vee [\![\psi_2]\!](\theta, ds, n) \\
[\![\psi_1\ \&\&\ \psi_2]\!](\theta, ds, n) & = & [\![\psi_1]\!](\theta, ds, n) \wedge [\![\psi_2]\!](\theta, ds, n) \\
[\![\psi_1\ =>\ \psi_2]\!](\theta, ds, n) & = & [\![\psi_1]\!](\theta, ds, n) \Rightarrow [\![\psi_2]\!](\theta, ds, n) \\
[\![\texttt{forall}\ X : \tau.\ \psi]\!](\theta, ds, n) & = & \forall t : \tau.\ \ [\![\psi]\!](\theta[X \mapsto t], ds, n) \\
[\![\texttt{exists}\ X : \tau.\ \psi]\!](\theta, ds, n) & = & \exists t : \tau.\ \ [\![\psi]\!](\theta[X \mapsto t], ds, n) \\
[\![t_1 == t_2]\!](\theta, ds, n) & = & [\![t_1]\!](\theta, n) = [\![t_2]\!](\theta, n) \\
[\![t_1\ !=\ t_2]\!](\theta, ds, n) & = & [\![t_1]\!](\theta, n) \neq [\![t_2]\!](\theta, n) \\
[\![EF(t_1, \ldots, t_i)@\texttt{in}[h]]\!](\theta, ds, n) & = & EF([\![t_1]\!](\theta, n), \ldots, [\![t_i]\!](\theta, n)) \in ds[h]
\end{array}$$

where the semantics of a term t at a node n under substitution θ is given by the function $[\![t]\!] : \mathit{Subst} \times \mathit{Node} \to \mathit{Term}$, which is defined as follows (where each primitive function f

has a meaning $[\![f]\!]$ *given in Definition 17):*

$$\begin{aligned}
[\![\texttt{currStmt}]\!](\theta, n) &= \mathit{stmtAt}(n) \\
[\![\texttt{currNode}]\!](\theta, n) &= n \\
[\![\mathit{Expr}]\!](\theta, n) &= \theta(\mathit{Expr}) \\
[\![\mathit{Stmt}]\!](\theta, n) &= \theta(\mathit{Stmt}) \\
[\![f(t_1, \ldots, t_i)]\!](\theta, n) &= [\![f]\!]([\![t_1]\!](\theta, n), \ldots, [\![t_i]\!](\theta, n))
\end{aligned}$$

Definition 17 *The meaning of primitive function symbols is given by:*

$[\![\texttt{evalExpr}]\!] : \mathit{State} \times \mathit{Expr} \rightharpoonup \mathit{Value}$
$[\![\texttt{evalExpr}]\!](\eta, e) = \eta(e)$

$[\![\texttt{newConst}]\!] : \mathbb{Z} \to \mathit{Expr}$
$[\![\texttt{newConst}]\!](i) = \mathit{literal\ representing\ integer\ } i$

$[\![\texttt{getConst}]\!] : \mathit{Expr} \rightharpoonup \mathbb{Z}$
$[\![\texttt{getConst}]\!](e) = \mathit{integer\ value\ of\ integer\ literal\ } e$

$[\![\texttt{applyUnaryOp}]\!] : \mathit{UnaryOp} \times \mathit{Const} \rightharpoonup \mathit{Const}$
$[\![\texttt{applyUnaryOp}]\!](op, a) = [\![op]\!](a)$

$[\![\texttt{applyBinaryOp}]\!] : \mathit{BinaryOp} \times \mathit{Const} \times \mathit{Const} \rightharpoonup \mathit{Const}$
$[\![\texttt{applyBinaryOp}]\!](op, a, b) = [\![op]\!](a, b)$

$[\![\texttt{min}]\!] : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$[\![\texttt{min}]\!](a, b) = a \quad \mathit{if}\ a < b$
$[\![\texttt{min}]\!](a, b) = b \quad \mathit{otherwise}$

$[\![\texttt{max}]\!] : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$[\![\texttt{max}]\!](a, b) = a \quad \mathit{if}\ a > b$
$[\![\texttt{max}]\!](a, b) = b \quad \mathit{otherwise}$

Section 2.1.3 described how Rhodium disallows negated edge facts, and as a result, the flow function induced by a set of Rhodium propagation rules is guaranteed to be monotonic. This is formalized in the following theorem about the monotonicity of F_O:

Theorem 6 (monotonicity of F_O) *If the syntactic form* ψ_1 `=>` ψ_2 *is disallowed, and the syntactic form* `!`ψ *is allowed only if* ψ *is an equality (i.e.* t_1 `==` t_2*), or an inequality (i.e.* t_1 `!=` t_2*) then* F_O *is monotonic.*

Theorem 6, a proof of which is given in Appendix C, can be used to implement a simple syntactic check that guarantees the monotonicity of F_O. For each rule, transform

the antecedent as follows: convert ψ_1 `=>` ψ_2 to `!`ψ_1 `||` ψ_2, and then push all the negations through conjunctions, disjunctions, existentials, and universals (in the standard way, using DeMorgan's law). If after this conversion all the antecedents are in the form required by Theorem 6, then F_O is guaranteed to be monotonic.

5.3.3 Abstraction

The abstraction function $\alpha : D_c \to D$ maps an element c of the concrete domain to the most precise element d of the abstract domain that conservatively approximates c. The meaning declarations provided by the Rhodium programmer implicitly define this abstraction function. In particular, given a set c of stores, $\alpha(c)$ returns the set of facts whose meanings hold of all stores in c. This is formalized in the following definition:

Definition 18 (forward abstraction function) *The forward abstraction function* $\alpha : D_c \to D$ *is defined as:*

$$\begin{aligned}\alpha(\eta s) = \{EF(t_1, \ldots, t_i) \mid\ & EF \in EdgeFact_O \wedge i = arity(EF) \wedge \\ & (t_1, \ldots, t_i) \in GroundTerm^i \wedge \\ & \forall \eta \in \eta s \ .\ [\![EF]\!](t_1, \ldots, t_i, \eta)\}\end{aligned} \tag{5.4}$$

In the above definition, $[\![EF]\!]$ stands for the meaning of fact schema EF, as defined below:

Definition 19 *The meaning of a forward-edge-fact-schema declaration:*

define forward edge fact $EF(X_1 : \tau_1, \ldots, X_i : \tau_i)$ **with meaning** M

is given by $[\![EF]\!] : \tau_1 \times \ldots \times \tau_i \times State \to bool$ *and is defined as:*

$$[\![EF]\!](t_1, \ldots, t_i, \eta) = [\![\theta(M)]\!](\eta)$$

where $\theta = [X_1 \mapsto t_1, \ldots, X_i \mapsto t_i]$, $\theta(M)$ *applies the substitution* θ *to* M, *and* $[\![M]\!] : State \to bool$ *evaluates the meaning predicate* M *on a given program state.*

Definition 20 *The evaluation function* $[\![M]\!] : State \to bool$ *for a meaning predicate* M *is defined as follows (where each primitive predicate* p *has a meaning* $[\![p]\!]$ *given in Defini-*

tion 21):

$$
\begin{array}{lcl}
[\![\texttt{true}]\!](\eta) & = & \mathit{true} \\
[\![\texttt{false}]\!](\eta) & = & \mathit{false} \\
[\![!\ M]\!](\eta) & = & \neg[\![M]\!](\eta) \\
[\![M_1\ ||\ M_2]\!](\eta) & = & [\![M_1]\!](\eta) \vee [\![M_2]\!](\eta) \\
[\![M_1\ \&\&\ M_2]\!](\eta) & = & [\![M_1]\!](\eta) \wedge [\![M_2]\!](\eta) \\
[\![M_1\ =>\ M_2]\!](\eta) & = & [\![M_1]\!](\eta) \Rightarrow [\![M_2]\!](\eta) \\
[\![\texttt{forall}\ X : \tau.\ M]\!](\eta) & = & \forall t : \tau.\ [\![\ [X \mapsto t](M)\ ]\!](\eta) \\
[\![\texttt{exists}\ X : \tau.\ M]\!](\eta) & = & \exists t : \tau.\ [\![\ [X \mapsto t](M)\ ]\!](\eta) \\
[\![t_1 == t_2]\!](\eta) & = & [\![t_1]\!](\eta) = [\![t_2]\!](\eta) \\
[\![t_1\ !=\ t_2]\!](\eta) & = & [\![t_1]\!](\eta) \neq [\![t_2]\!](\eta) \\
[\![p(t_1, \ldots, t_i)]\!](\eta) & = & [\![p]\!]([\![t_1]\!](\eta), \ldots, [\![t_i]\!](\eta))
\end{array}
$$

where the semantics of a term t is defined as (where each primitive function f has a meaning $[\![f]\!]$ *given in Definition 17):*

$$
\begin{array}{lcl}
[\![\texttt{eta}]\!](\eta) & = & \eta \\
[\![\mathit{Expr}]\!](\eta) & = & \mathit{Expr} \\
[\![\mathit{Stmt}]\!](\eta) & = & \mathit{Stmt} \\
[\![f(t_1, \ldots, t_i)]\!](\eta) & = & [\![f]\!]([\![t_1]\!](\eta), \ldots, [\![t_i]\!](\eta))
\end{array}
$$

There are no cases for terms η_1 or η_2 in the above definition of $[\![t]\!]$, since these two terms can only appear in the meaning of backward fact schemas.

Definition 21 *The meaning of primitive predicate symbols is given by (where*

numInEdges(s) *returns the number of input edges of a statement* s*):*

$$[\![\texttt{isStmtNotStuck}]\!] : \mathit{State} \times \mathit{Stmt} \to \mathit{bool}$$
$$[\![\texttt{isStmtNotStuck}]\!](\eta, s) =$$
$$\forall n \in \mathit{Node}, h \in [0 \mathinner{..} \mathit{numInEdges}(s) - 1] \;.$$
$$[\mathit{stmtAt}(n) = s] \Rightarrow \left[\exists (h', \eta') \in \mathbb{N} \times \mathit{State} \;.\; h, \eta \xrightarrow{n} h', \eta'\right]$$
$$[\![\texttt{isExprNotStuck}]\!] : \mathit{State} \times \mathit{Expr} \to \mathit{bool}$$
$$[\![\texttt{isExprNotStuck}]\!](\eta, e) = \exists v \in \mathit{Value} \;.\; \eta(e) = v$$

$$[\![\texttt{isLoc}]\!] : \mathit{Value} \to \mathit{bool}$$
$$[\![\texttt{isLoc}]\!](v) = v \in \mathit{Loc}$$

$$[\![\texttt{isConst}]\!] : \mathit{Value} \to \mathit{bool}$$
$$[\![\texttt{isConst}]\!](v) = v \in \mathit{Const}$$

$$[\![\texttt{equalUpTo}]\!] : \mathit{Var} \times \mathit{State} \times \mathit{State} \to \mathit{bool}$$
$$[\![\texttt{equalUpTo}]\!](x, (\rho_1, \sigma_1, \xi_1, \mathcal{M}_1), (\rho_2, \sigma_2, \xi_2, \mathcal{M}_2)) =$$
$$\rho_1 = \rho_2 \wedge \xi_1 = \xi_2 \wedge \mathcal{M}_1 = \mathcal{M}_2 \wedge \forall l \in \mathit{Loc} \;.\; [\rho_1(x) \neq l] \Rightarrow \sigma_1(l) = \sigma_2(l)$$

The forward abstraction function is join-monotonic, as required by the analysis framework from Chapter 4. This is stated in the following lemma, which is proved in Appendix C:

Lemma 2 *The forward abstraction function* α *from Definition 18 is join-monotonic.*

5.3.4 Replacement function

The replacement function $R_O : \mathit{Node} \times D^* \to \mathit{Graph} \cup \{\epsilon\}$ uses the transformation rules from O to determine whether or not to perform a transformation. In particular, $R_O(n, ds)$ returns a replacement graph if there is a transformation rule that triggers at node n given incoming facts ds, where the replacement graph contains a single node representing the transformed statement from the rule.[4] If no transformation rules trigger, then $R_O(n, ds)$ returns ϵ. If more than one transformation rule triggers, then $R_O(n, ds)$ nondeterministically chooses one of the transformations to return. This is formalized in the following definition:

Definition 22 (forward replacement function) *The replacement function* R_O *of*

[4]The replacement graph contains a single node even if the transformed statement is a conditional. Conditionals in the Rhodium IL do not contain sub-statements – they are simply guarded goto statements. A conditional is therefore represented in the CFG using a single branch node that has two successors.

$(\mathcal{A}_O, R_O)$ *is defined by:*

$$R_O(n, ds) = \begin{cases} singleNodeGraph(n, \theta(s)) & if \left[\begin{array}{l} (\textbf{if } \psi \textbf{ then transform to } s) \in O \wedge \\ [\![\psi]\!](\theta, ds, n) \end{array} \right] \\ \epsilon & otherwise \end{cases} \tag{5.5}$$

where $singleNodeGraph(n, s)$ *creates a sub-graph with a single node* n' *that satisfies* $stmtAt(n') = s$ *and that has as many input and output edges as* n*.*

5.4 Soundness checker

The Rhodium soundness checker uses a theorem prover to discharge a local soundness condition for each propagation and transformation rule. These conditions have already been presented informally in Section 2.2. I now present these conditions formally.

5.4.1 Propagation rules

Definition 23 *A forward propagation rule* **if** ψ **then** $EF(t_1, \ldots, t_i)@\texttt{out}[h']$ *is said to be sound iff the following condition holds:*

$$\forall (n, \eta, \eta', h, ds, \theta) \in Node \times State \times State \times \mathbb{N} \times D^* \times Subst.$$
$$\left[\begin{array}{l} [\![\psi]\!](\theta, ds, n) \wedge \\ h, \eta \overset{n}{\hookrightarrow} h', \eta' \wedge \\ allMeaningsHold(ds[h], \eta) \end{array} \right] \Rightarrow [\![EF]\!](\theta(t_1), \ldots, \theta(t_i), \eta') \tag{fwd-prop-sound}$$

where $allMeaningsHold$ *is defined as follows:*

$$allMeaningsHold(d, \eta) \triangleq \forall (EF, (t_1, \ldots, t_i)) \in EdgeFact_O \times GroundTerm^i.$$
$$EF(t_1, \ldots, t_i) \in d \Rightarrow [\![EF]\!](t_1, \ldots, t_i, \eta)$$

Intuitively, the above condition says the following. Suppose the given propagation rule fires at a node n on some incoming facts ds. Then, for every index h into ds and for every program state η such that (1) η on input edge h steps through n to η' on output edge h' and (2) the meanings of all the dataflow facts in $ds[h]$ hold of η, it should be the case that the meaning of the propagated dataflow fact holds of η'.

5.4.2 Transformation rules

Definition 24 *A forward transformation rule* **if** ψ **then transform to** s *is said to be sound iff the following condition holds:*

$$\forall(n, n', \eta, \eta', h, h', ds, \theta) \in \mathit{Node} \times \mathit{Node} \times \mathit{State} \times \mathit{State} \times \mathbb{N} \times \mathbb{N} \times D^* \times \mathit{Subst}.$$
$$\left[\begin{array}{l} [\![\psi]\!](\theta, ds, n) \wedge \\ h, \eta \overset{n}{\hookrightarrow} h', \eta' \wedge \\ \mathit{stmtAt}(n') = \theta(s) \wedge \\ \mathit{allMeaningsHold}(ds[h], \eta) \end{array}\right] \Rightarrow \quad h, \eta \overset{n'}{\hookrightarrow} h', \eta'$$

(fwd-trans-sound)

Intuitively, the above condition says the following. Suppose the given transformation rule fires at a node n on some incoming facts ds. Then, for every index h into ds and for every program state η such that (1) η on input edge h steps through n to η' on output edge h' and (2) the meanings of all the dataflow facts in $ds[h]$ hold of η, it should be the case that η on edge h also steps to η' on edge h' through the transformed node n'.

5.4.3 Soundness of the approach

The Rhodium strategy for checking soundness uses a theorem prover to discharge (fwd-prop-sound) and (fwd-trans-sound). For this strategy to be correct, it must be shown that if conditions (fwd-prop-sound) and (fwd-trans-sound) hold for all propagation rules and transformation rules of an optimization O, then the optimization O is sound. This is formalized in the following definition and theorem:

Definition 25 *We say that a forward Rhodium optimization O is sound iff $(\mathcal{A}_O, R_O)$ is sound (according to Definition 7).*

Theorem 7 (main forward soundness) *If all propagation rules and transformation rules in a forward Rhodium optimization O are sound, then O is sound.*

Theorem 7 follows trivially from the following two lemmas, which are proven in Appendix C, and Theorem 5:

Lemma 3 *If all propagation rules in a forward Rhodium optimization O are sound then F_O as defined in* (5.3) *is sound.*

Lemma 4 *If all transformation rules in a forward Rhodium program O are sound, then R_O as defined in* (5.5) *is sound.*

Furthermore, if all propagation rules of O are sound, then the induced analysis $\mathcal{A}_O$ is guaranteed to be sound, which can be useful if the computed facts are used for purposes other than optimizations. This is stated in the following theorem, which follows trivially from Lemma 3 and Theorem 4.

Theorem 8 (forward analysis soundness) *If all propagation rules in a forward Rhodium optimization O are sound, then the analysis $\mathcal{A}_O$ is sound.*

The above four theorems and lemmas (Theorems 7 and 8, and Lemmas 3 and 4) all appear in Figure 4.1, where they are all labeled "Forward".

5.4.4 Implementation

I have implemented the Rhodium strategy for automatically proving optimizations sound inside the Whirlwind compiler, using the Simplify automatic theorem prover [36]. The Whirlwind compiler supports a "verify_rhodium" command that runs the soundness checker on a Rhodium source file. For each propagation or transformation rule in the source file, the soundness checker asks Simplify to prove the soundness condition for that rule, given a set of background axioms. There are two kinds of background axioms: optimization-dependent ones and optimization-independent ones.

The optimization-dependent axioms encode the semantics of user-declared fact schemas and are generated automatically from the Rhodium source. In particular, for each edge-fact-schema definition, there is an axiom saying that if an instance of the fact schema appears on an input edge, then the meaning of the fact holds of the program state on that edge. These axioms together encode the *allMeaningsHold* assumptions from both (fwd-prop-sound) and (fwd-trans-sound). Furthermore, for each virtual-edge-fact definition and for each node-fact definition, there is an axiom stating that the fact being defined is equivalent to the body provided in the definition. Although the Rhodium formalization treats these facts as if they were expanded away, the implementation essentially lets the theorem prover perform this expansion, by providing axioms stating that a fact is equivalent to its defining body. The soundness checker generates these axioms by expanding case expressions

into ordinary boolean expressions and performing a few simple transformations to produce axioms in a form that the Simplify heuristics handle well. These transformations include merging consecutive quantifiers, pushing foralls into conjunctions (rewriting $\forall x.(A \wedge B)$ to $\forall x.A \wedge \forall x.B$), pushing foralls into the consequent (rewriting $\forall x.(A \Rightarrow B)$ to $A \Rightarrow \forall x.B$ if A does not use x) and pushing implications into conjunctions in the consequent (rewriting $A \Rightarrow (B \wedge C)$ to $(A \Rightarrow B) \wedge (A \Rightarrow C)$).

The optimization-independent axioms simply encode the semantics of the Rhodium intermediate language and they need not be modified in order to prove new optimizations sound. To encode the Rhodium intermediate language in Simplify, I introduce function symbols that represent term constructors for each kind of expression and statement. For example, the term $assgn(var(x), deref(var(y)))$ represents the statement $x := *y$. Next I formalize the representation of program states. Simplify has built-in axioms about a *map* data structure, with associated functions *select* and *update* to access elements and (functionally) update the map. This is useful for representing many components of a state. For example, an environment is a map from variables to locations, and a store is a map from locations to values.

Given the representation for states, I define axioms for a function symbol *evalExpr*, which evaluates an expression in a given state. The *evalExpr* function represents the function $\eta(\cdot)$ from Definition 1 (in Section 3.4). I also define axioms for a function *evalLExpr* which computes the location of a *lhs* expression given a program state. Then I provide axioms for the *stepEdge*, *stepEnv*, *stepStore*, *stepStack*, and *stepMem* functions, which together define the state transition function $h, \eta \xrightarrow{n} h', \eta'$ from Definition 2 (in Section 3.4). These functions take as input an edge index h, a state η, and a node n. The *stepEdge* function returns the resulting edge h', and the remaining functions return the various components of the stepped state η'. [5] As an example, the axioms for stepping an edge and a store through

[5] Instead of defining individual *stepEnv*, *stepStore*, *stepStack*, and *stepMem* functions, one could just as well have defined a single function *stepState* that returns a tuple.

an assignment $lhs := e$ are as follows:

$$\forall \eta, \pi, lhs, e.$$
$$\quad stmtAt(n) = assgn(lhs, e) \Rightarrow$$
$$\quad\quad stepEdge(h, \eta, n) = 0$$

$$\forall \eta, \pi, lhs, e.$$
$$\quad stmtAt(n) = assgn(lhs, e) \Rightarrow$$
$$\quad\quad stepStore(h, \eta, n) = update(store(\eta), evalLExpr(\eta, lhs), evalExpr(\eta, e))$$

The first axiom says that the resulting edge index is 0. The second axiom says that the new store is the same as the old one, but with the location of *lhs* updated to the value of e.

Finally, the $\hookrightarrow_\pi$ function is defined in terms of the $\rightarrow_\pi$ function. In the context of intraprocedural analysis, the bodies of called procedures are not accessible. Therefore, I conservatively model the semantics of stepping over a procedure call by a set of axioms that hold for *any* call. The primary axiom says that the store after a call preserves the values of local variables in the caller whose locations are not pointed to before the call. This axiom encodes the fact that locals not reachable from the store cannot be modified by a call.

The handwritten background axioms, which encode the semantics of the Rhodium intermediate language, comprise of about 3,000 lines of code written in Simplify's input language. For the current Rhodium code base of 29 edge-fact schemas, 19 node-fact schemas, 1 virtual-fact schema, 105 propagation rules, and 14 transformation rules, the automatically generated background axioms, encoding the semantics of edge facts, node facts and virtual edge facts, constitute about 7,000 lines of Simplify code.

Chapter 6

BACKWARD RHODIUM OPTIMIZATIONS

This chapter presents the formal details of how backward Rhodium optimizations are checked for soundness. The structure of this chapter is the same as Chapter 5, except that there is no section on syntax, since Section 5.1 already covered the entire syntax of Rhodium, including backward optimizations. Section 6.1 defines the concrete backward semantics, Section 6.2 presents the formal semantics of backward Rhodium optimizations, and finally Section 6.3 describes the details of how the soundness checker works for backward optimizations.

6.1 Concrete semantics

This section describes the backward concrete analysis $\mathcal{C}$ used to instantiate the framework from Chapter 4 for backward Rhodium optimizations. I have chosen the backward concrete semantics to be a backward state collecting semantics. This means that the concrete analysis computes, for each edge in a procedure's CFG, the set of states that the procedure can be in at that edge, given some final set of states that the procedure must be in after it has finished executing. The concrete flow function simply finds all possible states on the incoming edges of a statement that can step to a given set of states on the outgoing edges. These ideas are made precise in the following definition of the backward concrete analysis:

Definition 26 (backward concrete analysis) *The backward concrete analysis* $\mathcal{C} = (D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c, id, F_c)$ *is defined by:*

$$(D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c) = (2^{State}, \cup, \cap, \subseteq, State, \emptyset)$$

and

$$F_c(n, cs)[h] = \{\eta \mid \exists\ \eta' \in State, i \in \mathbb{N}\ .\ [\eta' \in cs[i]\ \ \wedge\ \ h, \eta \overset{n}{\hookrightarrow} i, \eta']\} \tag{6.1}$$

Although it may at first seem clumsy to have a backward concrete semantics that is different from the forward concrete semantics, this approach is standard in abstract inter-

pretation, and was originally presented by Cousot and Cousot in their seminal 1979 abstract interpretation paper (see section 3.2 of [30]). The advantage of defining a separate concrete semantics for backward optimizations is that, in this way, the backward semantics can be defined using backward flow functions, similar to the way that backward dataflow analyses are defined. As a result, both the concrete and the abstract semantics operate in the same direction, and the proofs of soundness become much simpler. On the other hand, if one uses a forward concrete semantics to prove the soundness of backward optimizations, the proofs become complex and subtle because of the inherent mismatch of a forward concrete semantics and a backward abstract semantics. In fact, the formalization of backward optimizations in Cobalt [64], Rhodium's predecessor, used a forward concrete semantics with a backward abstract semantics, and those proofs of soundness were much more complex than Rhodium's proofs.

The one issue with using a backward concrete semantics is that in the end, one really cares about preserving forward behavior. Unfortunately, preserving the backward collecting semantics does *not* guarantee that the forward collecting semantics is preserved *at each edge*. In fact, one of they key properties of a backward optimization is that there are transient periods during execution where the states in the original and the transformed programs are not at all the same. As a example, in dead assignment elimination, the states following the removed assignment are different in the original and the transformed program.

It is however the case that preserving the backward state collecting semantics of a procedure guarantees that its forward input-state-to-output-state behavior is preserved, for input states that do not cause the execution to get stuck or to loop forever. In particular, suppose that some original procedure starts in a state η, and terminates without getting stuck in state η'. The backward collecting semantics, if started with $\{\eta'\}$ at the end of this procedure, will compute the set S of states at the beginning of the procedure that will lead to η'. The set S must contain η, simply by the definition of the backward collecting semantics. If the original procedure is transformed in a way that preserves its backward collecting semantics, then the backward semantics, if started with $\{\eta'\}$ at the end of the transformed procedure, will compute a set S' at the beginning of the procedure that will satisfy $S' \supseteq S$. Since η belongs to S, it belongs to S' too, and this means, by the definition

of the backward semantics, that execution of the transformed procedure starting in η will terminate without getting stuck in η'.

For non-terminating runs, preservation of the backward collecting semantics provides a rather meaningless guarantee that is exactly the opposite of what one would really want. In particular, it makes no guarantees about the part of the run before the infinite loop, but requires the unreachable part of the procedure, after the infinite loop, to be preserved. This means, for example, that a backward optimization could transform an infinite loop into a program that gets stuck. This is an unfortunate side-effect of looking at the procedure from its endpoint. There may be ways of adapting the backward collecting semantics to address this limitation, for example by incorporating some sort of forward reachability information in the semantics.

Throughout the remainder of this chapter, it will sometimes be helpful to think of sets of states in the backward concrete analysis as predicates. In particular, a set of states H can be interpreted as a predicate on states $P : \mathit{State} \to \mathit{bool}$, where $P(\eta) \triangleq \eta \in H$. In this interpretation, the concrete backward semantics is a weakest precondition semantics, which computes, at each node in the CFG, the weakest condition that must hold at that edge for some final condition to hold at the end of the procedure. The backward concrete flow function is then just a weakest precondition computation. This weakest precondition semantics is exactly the one used by Cousot and Cousot in [30].[1]

Along with the concrete analysis, one must also specify the *callerToCallee* and *calleeToCaller* functions, which, for the backward case, are defined as follows:

Definition 27 (backward *callerToCallee*) *The backward version of callerToCallee is defined by:*

$$\mathit{callerToCallee}(n, cs) = cs$$

Definition 28 (backward *calleeToCaller*) *The backward version of calleeToCaller is de-*

[1]The weakest precondition mentioned here is *strict*, meaning that the precondition must also ensure that the statement terminates. In contrast, the weakest *liberal* precondition is not required to ensure that the statement terminates – it must only ensure that *if* the statement terminates, *then* the postcondition holds. Using liberal preconditions instead of strict ones would *not* solve the previously mentioned limitation that backward collecting semantics cannot make meaningful guarantees about non-terminating runs.

fined by:

$$calleeToCaller(n, cs)[0] = \{\eta \mid \exists\ \eta' \in State\ .\ [\eta' \in cs[0]\ \wedge\ 0, \eta \xrightarrow{cn} 1, \eta']\}$$
$$where\ cn = interCallNode(n)$$

It is easy to see from the above definitions that *calleeToCaller* and *callerToCallee* satisfy requirements (4.1), (4.2), (4.3), and (4.4). Furthermore, the concrete backward flow function defined in (6.1) computes the same information at call nodes as the flow function from Definition 5 in Section 4.1. Indeed, Equation (6.1) says that the set of states before a call node is computed by taking the union over the outgoing states of the corresponding incoming state. Definition 5 describes the same computation, except that it uses the collecting semantics of the callee to determine the set of all incoming states.

6.2 Rhodium semantics

The semantics of a backward Rhodium optimization, which is an optimization containing only backward rules, is defined analogously to the forward case, by using an associated AT-analysis. I fix the backward Rhodium optimization O whose semantics is being defined, and I define the semantics of O using an associated AT-analysis $(\mathcal{A}_O, R_O)$, where $\mathcal{A}_O = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, F_O, \alpha)$. I first define the lattice $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$, followed by the flow function F_O, the abstraction function α, and finally the replacement function R_O.

As in the forward case, I assume that all node facts and virtual edge facts in O have been macro-expanded away, I use $EdgeFact_O$ to denote the set of edge-fact schema names declared in O, and I use $r \in O$ to say that rule r appears in O.

6.2.1 Lattice

Similarly to the forward case, the lattice of the backward Rhodium AT-analysis is the power-set lattice of all backward Rhodium facts. The definition of this lattice is exactly the same as in Definition 14.

6.2.2 Flow function

The backward flow function is defined as in the forward case, except that `out` is replaced with `in`, and vice-versa. To be explicit, the backward flow function F_O is defined as follows:

Definition 29 (backward flow function) *The flow function F_O of $(\mathcal{A}_O, R_O)$ is defined by:*

$$F_O(n, ds)[h] = \{\theta(EF(t_1, \ldots, t_i)) \mid \begin{array}{l}(\textbf{if } \psi \textbf{ then } EF(t_1, \ldots, t_i)@\texttt{in}[h]) \in O \wedge \\ [\![\psi]\!]_b(\theta, ds, n)\}\end{array} \tag{6.2}$$

Definition 30 *The meaning function $[\![\psi]\!]_b : Subst \times D^* \times Node \to bool$ for the antecedent ψ of a backward rule is defined as follows:*

$$\begin{array}{lcl}
[\![\texttt{true}]\!]_b(\theta, ds, n) & = & \mathit{true} \\
[\![\texttt{false}]\!]_b(\theta, ds, n) & = & \mathit{false} \\
[\![!\ \psi]\!]_b(\theta, ds, n) & = & \neg[\![\psi]\!]_b(\theta, ds, n) \\
[\![\psi_1\ ||\ \psi_2]\!]_b(\theta, ds, n) & = & [\![\psi_1]\!]_b(\theta, ds, n) \vee [\![\psi_2]\!]_b(\theta, ds, n) \\
[\![\psi_1\ \&\&\ \psi_2]\!]_b(\theta, ds, n) & = & [\![\psi_1]\!]_b(\theta, ds, n) \wedge [\![\psi_2]\!]_b(\theta, ds, n) \\
[\![\psi_1 => \psi_2]\!]_b(\theta, ds, n) & = & [\![\psi_1]\!]_b(\theta, ds, n) \Rightarrow [\![\psi_2]\!]_b(\theta, ds, n) \\
[\![\texttt{forall}\ X : \tau.\ \psi]\!]_b(\theta, ds, n) & = & \forall t : \tau.\ \ [\![\psi]\!]_b(\theta[X \mapsto t], ds, n) \\
[\![\texttt{exists}\ X : \tau.\ \psi]\!]_b(\theta, ds, n) & = & \exists t : \tau.\ \ [\![\psi]\!]_b(\theta[X \mapsto t], ds, n) \\
[\![t_1 == t_2]\!]_b(\theta, ds, n) & = & [\![t_1]\!](\theta, n) = [\![t_2]\!](\theta, n) \\
[\![t_1\ !\!= t_2]\!]_b(\theta, ds, n) & = & [\![t_1]\!](\theta, n) \neq [\![t_2]\!](\theta, n) \\
[\![EF(t_1, \ldots, t_i)@\texttt{out}[h]]\!]_b(\theta, ds, n) & = & EF([\![t_1]\!](\theta, n), \ldots, [\![t_i]\!](\theta, n)) \in ds[h]
\end{array}$$

where the semantics of a term t at a node n under substitution θ is given by the function $[\![t]\!] : Subst \times Node \to Term$ from Definition 16.

The monotonicity theorem from Section 5.3.2 (Theorem 6) also holds of the backward flow function F_O.

6.2.3 Abstraction

As in the forward case, the meaning declarations provided by the programmer implicitly define the abstraction function α. However, in the backward case, there are two kinds of meanings that the programmer can declare: predicate meanings, and relational meanings.

A predicate meaning is a predicate over program states η, with the intuition being the same as in the forward case: if the fact appears on an edge in the CFG, then its meaning must hold of all program states computed by the concrete semantics at that edge. Figure 6.2.3 shows an example of a backward fact schema whose meaning is a predicate.

1. **define backward edge fact** *is_expr_not_stuck*(*E*:*Expr*)
2. **with meaning** `isExprNotStuck`(η, E)
3. **if** $stmt(X := E)$
4. **then** *is_expr_not_stuck*(X)@*in*
5. **if** $stmt(X := E)$
6. **then** *is_expr_not_stuck*(E)@*in*
7. **if** *is_expr_not_stuck*(E)@*out* $\wedge$ *conservUnchanged*(E)
8. **then** *is_expr_not_stuck*(E)@*in*

Figure 6.1: Example of a backward predicate meaning

The *is_expr_not_stuck*(E) schema captures the fact that an expression E is used downstream in the program without first being modified, and as a result must evaluate without getting stuck.[2] The `isExprNotStuck` primitive, which is provided by the system, returns true iff a given expression E evaluates without getting stuck in a given program state η. The first two rules, on lines 3–6, state that before the statement $X := E$, both X and E will be known to evaluate without getting stuck. The last rule, on lines 7–8, says that if an expression evaluates without getting stuck after a statement, and the expression is not modified by the statement, then it also evaluates without getting stuck before the statement. The *conservUnchanged*(E) node fact is a conservative version of *unchanged*(E). In particular, *conservUnchanged* does not use pointer information to determine whether or not the value of E changes, since pointer information is encoded using forward facts, and such forward facts cannot be used in backward rules.

Unfortunately, there are some useful backward fact schemas whose meanings cannot be expressed as predicates. Consider, for example, the *dead*(X) fact schema. Intuitively, the presence of *dead*(X) on a CFG edge e means that the value of X at edge e does not affect the future behavior of the program. Unfortunately, this cannot be expressed as a predicate that holds of all program states at edge e, because it requires mentioning the future behavior

[2]Recall from Section 4.2 that the Rhodium strategy for proving soundness preserves the behavior of those runs of the original program that do not get stuck.

of the program. Rhodium therefore supports a second kind of backward meaning, called a relational meaning. Instead of a predicate on program states, a relational meaning is a relation over two program states η_1 and η_2. The intuition is that if a backward fact appears on an edge, and the fact's relational meaning holds of two program states η_1 and η_2, then η_1 and η_2 should produce the same future behavior of the program, from that edge onward.

As an example, consider the relational meaning of the $dead(X)$ fact schema from Figure 2.5, which says that η_1 and η_2 are equal up to X. The notation $\eta_1/X = \eta_2/X$ is mathematical sugar for `equalUpTo`(X, η_1, η_2), where `equalUpTo` is a primitive provided by the Rhodium system. The meaning of $dead(X)$ therefore says that if a variable X is identified as being dead, then it better be the case that if η_1 and η_2 agree on all variables except X, then η_1 and η_2 will lead to the same future behavior of the program. In other words, the value of a dead variable X does not affect the future behavior of the program, which captures exactly the intended semantics of dead variables.

The system determines whether a backward meaning is a predicate or a relation by looking at its structure: if the meaning mentions η, then it is a predicate meaning (and therefore cannot mention η_1 or η_2); if it mentions η_1 or η_2, then it is a relational meaning (and therefore cannot mention η); if it contains no mention of η, η_1 or η_2, then it is considered a predicate meaning (although it could also be considered a relational meaning).

The above ideas are now made more precise. I use $EdgeFact_p$ to denote the set of fact-schema names declared in O with a predicate meaning, and I call such fact schemas "backward-predicate-fact schemas". Similarly, I use $EdgeFact_r$ to denote the set of fact-schema names declared in O with a relational meaning, and I call such fact schemas "backward-relational-fact schemas". As an example, $is\_expr\_not\_stuck(E)$ is a predicate-fact schema, and so $is\_expr\_not\_stuck \in EdgeFact_p$, while $dead(X)$ is a relational-fact schema, and so $dead \in EdgeFact_r$. Furthermore, a rule that propagates a predicate fact is called a predicate rule, and one that propagates a relational fact is a relational rule. Predicate rules can use only predicate facts, relational rules can use only relational facts, and transformation rules are allowed to use both predicate and relational facts.

The backward abstraction function $\alpha : D_c \to D$ is defined as follows:

Definition 31 (backward abstraction function) *The backward abstraction function* $\alpha : D_c \to D$ *is defined as:*

$$\begin{aligned}\alpha(\eta s) = \{EF(t_1, \ldots, t_i) \mid \quad & EF \in EdgeFact_O \wedge i = arity(EF) \wedge \\ & (t_1, \ldots, t_i) \in GroundTerm^i \wedge \\ & [\![EF]\!](t_1, \ldots, t_i, \eta s)\}\end{aligned} \tag{6.3}$$

In the above definition, $[\![EF]\!]$ stands for the meaning of fact schema EF. In contrast to the forward case, where $[\![EF]\!]$ looked only at an individual program state $\eta \in \eta s$, in the backward case, $[\![EF]\!]$ operates on the whole set of states ηs. Depending on whether EF is a predicate-fact schema or a relational-fact schema, $[\![EF]\!]$ is defined differently. For predicate-fact schemas, $[\![EF]\!]$ simply makes sure that the meaning declared by the programmer holds of all program states in ηs, as stated in the following definition:

Definition 32 *The meaning of a backward-predicate-fact-schema declaration:*

define backward edge fact $EF(X_1 : \tau_1, \ldots, X_i : \tau_i)$ **with meaning** M

is given by $[\![EF]\!] : \tau_1 \times \ldots \times \tau_i \times 2^{State} \to bool$ *and is defined as:*

$$[\![EF]\!](t_1, \ldots, t_i, \eta s) = \forall \eta \in \eta s \; . \; [\![EF]\!]_p(t_1, \ldots, t_i, \eta)$$

where $[\![EF]\!]_p : \tau_1 \times \ldots \times \tau_i \times State \to bool$ *is defined as:*

$$[\![EF]\!]_p(t_1, \ldots, t_i, \eta) = [\![\theta(M)]\!](\eta)$$

where $\theta = [X_1 \mapsto t_1, \ldots, X_i \mapsto t_i]$, $\theta(M)$ *applies the substitution* θ *to* M, *and* $[\![M]\!] : State \to bool$ *is the evaluation function from Definition 20.*

The $[\![EF]\!]$ function for relational-fact schemas is more complicated to understand. I present its definition first, followed by some explanations.

Definition 33 *The meaning of a backward-relational-fact-schema declaration:*

define backward edge fact $EF(X_1 : \tau_1, \ldots, X_i : \tau_i)$ **with meaning** M

is given by $[\![EF]\!] : \tau_1 \times \ldots \times \tau_i \times 2^{State} \to bool$ *and is defined as:*

$$\begin{aligned}[\![EF]\!](t_1, \ldots, t_i, \eta s) = \quad & \forall (\eta_1, \eta_2) \in State \times State \; . \\ & [\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (\eta_1 \in \eta s \Leftrightarrow \eta_2 \in \eta s)\end{aligned}$$

where $[\![EF]\!]_r : \tau_1 \times \ldots \times \tau_i \times State \times State \rightarrow bool$ *is defined as:*

$$[\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) = [\![\theta(M)]\!]_r(\eta_1, \eta_2)$$

where $\theta = [X_1 \mapsto t_1, \ldots, X_i \mapsto t_i]$, $\theta(M)$ *applies the substitution* θ *to* M, *and* $[\![M]\!]_r : State \times State \rightarrow bool$ *evaluates the relation* M *on two given program states.*

Definition 34 *The evaluation function* $[\![M]\!]_r : State \times State \rightarrow bool$ *for a relational meaning* M *is defined as follows (where each primitive predicate* p *has a meaning* $[\![p]\!]$ *and each primitive function* f *has a meaning* $[\![f]\!]$*):*

$$\begin{array}{lcl}
[\![\texttt{true}]\!]_r(\eta_1, \eta_2) & = & true \\
[\![\texttt{false}]\!]_r(\eta_1, \eta_2) & = & false \\
[\![!\ M]\!]_r(\eta_1, \eta_2) & = & \neg [\![M]\!]_r(\eta_1, \eta_2) \\
[\![M_1\ ||\ M_2]\!]_r(\eta_1, \eta_2) & = & [\![M_1]\!]_r(\eta_1, \eta_2) \vee [\![M_2]\!]_r(\eta_1, \eta_2) \\
[\![M_1\ \&\&\ M_2]\!]_r(\eta_1, \eta_2) & = & [\![M_1]\!]_r(\eta_1, \eta_2) \wedge [\![M_2]\!]_r(\eta_1, \eta_2) \\
[\![M_1 => M_2]\!]_r(\eta_1, \eta_2) & = & [\![M_1]\!]_r(\eta_1, \eta_2) \Rightarrow [\![M_2]\!]_r(\eta_1, \eta_2) \\
[\![\texttt{forall}\ X : \tau.\ M]\!]_r(\eta_1, \eta_2) & = & \forall t : \tau.\ [\![\ [X \mapsto t](M)\ ]\!]_r(\eta_1, \eta_2) \\
[\![\texttt{exists}\ X : \tau.\ M]\!]_r(\eta_1, \eta_2) & = & \exists t : \tau.\ [\![\ [X \mapsto t](M)\ ]\!]_r(\eta_1, \eta_2) \\
[\![t_1 == t_2]\!]_r(\eta_1, \eta_2) & = & [\![t_1]\!]_r(\eta_1, \eta_2) = [\![t_2]\!]_r(\eta_1, \eta_2) \\
[\![t_1\ != t_2]\!]_r(\eta_1, \eta_2) & \neq & [\![t_1]\!]_r(\eta_1, \eta_2) = [\![t_2]\!]_r(\eta_1, \eta_2) \\
[\![p(t_1, \ldots, t_i)]\!]_r(\eta_1, \eta_2) & = & [\![p]\!]([\![t_1]\!]_r(\eta_1, \eta_2), \ldots, [\![t_i]\!]_r(\eta_1, \eta_2))
\end{array}$$

where the relation semantics of a term t *is defined as (where each primitive function* f *has a meaning* $[\![f]\!]$ *given in Definition 17):*

$$\begin{array}{lcl}
[\![\texttt{eta_1}]\!]_r(\eta_1, \eta_2) & = & \eta_1 \\
[\![\texttt{eta_2}]\!]_r(\eta_1, \eta_2) & = & \eta_2 \\
[\![Expr]\!]_r(\eta_1, \eta_2) & = & Expr \\
[\![Stmt]\!]_r(\eta_1, \eta_2) & = & Stmt \\
[\![f(t_1, \ldots, t_i)]\!]_r(\eta_1, \eta_2) & = & [\![f]\!]([\![t_1]\!]_r(\eta_1, \eta_2), \ldots, [\![t_i]\!]_r(\eta_1, \eta_2))
\end{array}$$

Definition 33 at first seems daunting. However, by interpreting sets as predicates, the definition becomes easier to understand. In particular, if the set ηs is interpreted as a weakest precondition P computed by the concrete semantics, then the following part of the definition:

$$\begin{array}{lll}
[\![EF]\!](t_1, \ldots, t_i, \eta s) = & \forall (\eta_1, \eta_2) \in State \times State\ . & \\
 & [\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (\eta_1 \in \eta s \Leftrightarrow \eta_2 \in \eta s) &
\end{array}$$

becomes:

$$[\![EF]\!](t_1, \ldots, t_i, P) = \quad \forall(\eta_1, \eta_2) \in \mathit{State} \times \mathit{State} \;.\; [\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (P(\eta_1) \Leftrightarrow P(\eta_2))$$

In this interpretation, if a relational fact appears on a CFG edge e, and if its relational meaning holds of η_1 and η_2, then the weakest precondition computed by the concrete semantics at e will have the same truth value when evaluated in η_1 and in η_2. This means that η_1 and η_2 are indistinguishable when it comes to establishing a given postcondition at the end of the procedure: either they both establish the postcondition, or they both don't. This exactly captures the idea that η_1 and η_2 should lead to the same future behavior of the program.

As an example, consider the meaning of the $\mathit{dead}(X)$ fact schema, $\eta_1/X = \eta_2/X$, while still using the sets-as-predicates interpretation. The $[\![\mathit{dead}]\!]$ function then becomes:

$$[\![\mathit{dead}]\!](X, P) = \forall(\eta_1, \eta_2) \in \mathit{State} \times \mathit{State} \;.\; \eta_1/X = \eta_2/X \Rightarrow (P(\eta_1) \Leftrightarrow P(\eta_2))$$

In this case, the body of $[\![\mathit{dead}]\!]$ says that the predicate P evaluates to the same truth value on any two states that differ only on X, which is the same as saying that P *does not depend on* X. As a result, the semantics of $\mathit{dead}(X)$ appearing on a CFG edge is that the weakest precondition at that edge does not depend on X. This exactly captures the fact that the value of X does not affect the future behavior of the program.

As required by the analysis framework from Chapter 4, the backward abstraction function is join-monotonic. This is stated in the following lemma, which is proved in Appendix D:

Lemma 5 *The backward abstraction function α from Definition 31 is join-monotonic.*

6.2.4 Replacement function

The replacement function R_O for the backward case is similar to the forward case, except that it uses $[\![\cdot]\!]_b$ instead of $[\![\cdot]\!]$:

Definition 35 (backward replacement function) *The replacement function R_O of*

$(\mathcal{A}_O, R_O)$ *is defined by:*

$$R_O(n, ds) = \begin{cases} singleNodeGraph(n, \theta(s)) & if \left[\begin{array}{l} (\textbf{if } \psi \textbf{ then transform to } s) \in O \wedge \\ [\![\psi]\!]_b(\theta, ds, n) \end{array} \right] \\ \epsilon & otherwise \end{cases} \tag{6.4}$$

where $singleNodeGraph(n, s)$ *creates a sub-graph with a single node* n' *that satisfies* $stmtAt(n') = s$ *and that has as many input and output edges as* n*.*

6.3 Soundness checker

This section describes the soundness conditions for backward propagation and transformation rules, and how these soundness conditions are implemented in the Simplify theorem prover.

6.3.1 Propagation rules

Definition 36 *A backward predicate propagation rule* **if** ψ **then** $EF(t_1, \ldots, t_i)@\texttt{in}[h]$ *is said to be sound iff the following condition holds:*

$$\forall (n, \eta, \eta', h', ds, \theta) \in Node \times State \times State \times \mathbb{N} \times D^* \times Subst.$$
$$\left[\begin{array}{l} [\![\psi]\!]_b(\theta, ds, n) \wedge \\ h, \eta \overset{n}{\hookrightarrow} h', \eta' \wedge \\ allMeaningsHold_p(ds[h'], \eta') \end{array} \right] \Rightarrow [\![EF]\!]_p(\theta(t_1), \ldots, \theta(t_i), \eta)$$

(bwd-prop-sound-pr)

where $allMeaningsHold_p$ *is defined as follows:*

$$allMeaningsHold_p(d, \eta) \triangleq \forall (EF, (t_1, \ldots, t_i)) \in EdgeFact_p \times GroundTerm^i.$$
$$EF(t_1, \ldots, t_i) \in d \Rightarrow [\![EF]\!]_p(t_1, \ldots, t_i, \eta)$$

The above condition is analogous to (fwd-prop-sound), except that facts in the antecedent provide information about the outgoing program state η' and the meaning of the propagated fact has to be shown on the incoming program state η.

Definition 37 *A backward relational propagation rule* **if** ψ **then** $EF(t_1, \ldots, t_i)@\texttt{in}[h]$ *is said to be sound iff the following two conditions hold:*

$$\forall(n, \eta_1, \eta_2, h', ds, \theta) \in \mathit{Node} \times \mathit{State} \times \mathit{State} \times \mathbb{N} \times D^* \times \mathit{Subst}.$$
$$\left[\begin{array}{l} [\![EF]\!]_r(\theta(t_1), \ldots, \theta(t_i), \eta_1, \eta_2) \wedge \\ [\![\psi]\!]_b(\theta, ds, n) \end{array}\right] \Rightarrow \left[\begin{array}{l} \exists \eta_1' \in \mathit{State} \;.\; h, \eta_1 \overset{n}{\hookrightarrow} h', \eta_1' \;\Leftrightarrow \\ \exists \eta_2' \in \mathit{State} \;.\; h, \eta_2 \overset{n}{\hookrightarrow} h', \eta_2' \end{array}\right]$$

(bwd-prop-sound-rel-1)

$$\forall(n, \eta_1, \eta_2, \eta_1', \eta_2', h', ds, \theta) \in \mathit{Node} \times \mathit{State} \times \mathit{State} \times \mathit{State} \times \mathit{State} \times \mathbb{N} \times D^* \times \mathit{Subst}.$$
$$\left[\begin{array}{l} [\![EF]\!]_r(\theta(t_1), \ldots, \theta(t_i), \eta_1, \eta_2) \wedge \\ [\![\psi]\!]_b(\theta, ds, n) \wedge \\ h, \eta_1 \overset{n}{\hookrightarrow} h', \eta_1' \wedge \\ h, \eta_2 \overset{n}{\hookrightarrow} h', \eta_2' \end{array}\right] \Rightarrow \left[\begin{array}{l} \eta_1' = \eta_2' \vee \\ \mathit{someMeaningHolds}(ds[h'], \eta_1', \eta_2') \end{array}\right]$$

(bwd-prop-sound-rel-2)

where someMeaningHolds is defined as follows:

$$\begin{aligned} \mathit{someMeaningHolds}(d, \eta_1, \eta_2) \triangleq \;& \exists (EF, (t_1, \ldots, t_i)) \in \mathit{EdgeFact}_r \times \mathit{GroundTerm}^i. \\ & EF(t_1, \ldots, t_i) \in d \wedge [\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \end{aligned}$$

The intuition behind the above two conditions is as follows. To show that the EF fact can correctly be propagated to $\texttt{in}[h]$, one must show that if the relational meaning of EF holds of two states η_1 and η_2, then η_1 and η_2 will lead to the same future behavior of the program, from $\texttt{in}[h]$ onward. This can be achieved by guaranteeing that (1) η_1 and η_2 step to the same edge $\texttt{out}[h']$, producing program states η_1' and η_2', and (2) that η_1' and η_2' lead to the same future behavior of the program, but this time from $\texttt{out}[h']$ onward. These are exactly the two conditions (bwd-prop-sound-rel-1) and (bwd-prop-sound-rel-2). Condition (bwd-prop-sound-rel-1) guarantees if both η_1 and η_2 step, then they reach the same outgoing edge: either both η_1 and η_2 get stuck, or they both step to the same outgoing edge. Condition (bwd-prop-sound-rel-2) guarantees that if η_1 and η_2 step to η_1' and η_2', then η_1' and η_2' will lead to the same future behavior of the program, from $\texttt{out}[h']$ onward. One way of achieving this is to have η_1' and η_2' be equal. Another way is to have some backward relational fact EF' be present on $\texttt{out}[h']$, with the relational meaning of EF' holding of η_1'

and η_2'. In this case, the semantics of EF' implies that η_1' and η_2' will lead to the same future behavior of the program, from $\texttt{out}[h']$ onward.

As an example, consider the following backward rule for dead assignment elimination, from lines 4–5 of Figure 2.5:

if $stmt(X := E) \wedge mustNotUse(X)$
then $dead(X)@in$

This rule is sound because a statement $X := E$ that does not use X produces the same outgoing state when executed on two incoming states that are equal up to X. For this rule, it is the $\eta_1 = \eta_2$ case of condition (bwd-prop-sound-rel-2) that holds. As another example, consider the backward rule from lines 8–9 of Figure 2.5:

if $dead(X)@out \wedge mustNotUse(X)$
then $dead(X)@in$

This rule is sound because a statement that does not use X, when executed on two incoming states that are equal up to X, produces two states that are also equal up to X. For this rule, it is the *someMeaningHolds* case of (bwd-prop-sound-rel-2) that holds, where the existential in *someMeaningHolds* is valid because the $dead(X)$ fact belongs to the dataflow information $ds[h']$, which follows from the $dead(X)@out$ conjunct in the rule's antecedent ψ, combined with the $[\![\psi]\!]_b(\theta, ds, n)$ assumption in (bwd-prop-sound-rel-2).

6.3.2 Transformation rules

Definition 38 *A backward transformation rule* **if** ψ **then transform to** s *is said to be sound iff the following condition holds:*

$$\forall (n, n', \eta, \eta', h, h', ds, \theta) \in Node \times Node \times State \times State \times \mathbb{N} \times \mathbb{N} \times D^* \times Subst.$$
$$\left[\begin{array}{l} [\![\psi]\!]_b(\theta, ds, n) \wedge \\ h, \eta \overset{n}{\hookrightarrow} h', \eta' \wedge \\ stmtAt(n') = \theta(s) \wedge \\ allMeaningsHold_p(ds[h'], \eta') \end{array}\right] \Rightarrow \exists \eta'' \in State\ .\ \left[\begin{array}{l} h, \eta \overset{n'}{\hookrightarrow} h', \eta'' \wedge \\ (\ \eta' = \eta'' \vee \\ \quad someMeaningHolds(ds[h'], \eta', \eta'')\ ) \end{array}\right]$$

(bwd-trans-sound)

Intuitively, the above condition works as follows. To show that node n can correctly be transformed to node n', one must show that the future behavior of the program starting in a state η will be the same going through n and n'. This can be achieved by showing that the states η' and η'', produced by stepping η through n and n', lead to the same future behavior of the program. This, in turn, can be achieved by either having η' and η'' be equal, or by having some backward relational fact be present on $\texttt{out}[h']$, with the relational meaning of the fact holding of η' and η''.

As an example, consider the transformation rule for dead assignment elimination, from lines 10–11 of Figure 2.5:

if $stmt(X := E) \wedge dead(X)@out$
then transform to `skip`

This rule is sound because when the statements $X := E$ and `skip` are executed in a program state η, the resulting program states η' and η'' are equal up to X. For this rule, it is the *someMeaningHolds* case of (bwd-trans-sound) that holds, where the existential in *someMeaningHolds* is valid because the $dead(X)$ fact belongs to the dataflow information $ds[h']$, which follows from the $dead(X)@out$ conjunct in the rule's antecedent ψ, combined with the $[\![\psi]\!]_b(\theta, ds, n)$ assumption in (bwd-trans-sound).

6.3.3 Soundness of the approach

Similarly to the forward case, it must be shown that, for a backward optimization, if all predicate rules satisfy (bwd-prop-sound-pr), all relational rules satisfy (bwd-prop-sound-rel-1) and (bwd-prop-sound-rel-2), and all transformation rules satisfy (bwd-trans-sound), then the backward optimization is sound. This is formalized in the following definition and theorem:

Definition 39 *We say that a backward Rhodium optimization O is sound iff $(\mathcal{A}_O, R_O)$ is sound (according to Definition 7)*

Theorem 9 (main backward soundness) *If all propagation rules and transformation rules in a backward Rhodium optimization O are sound, then O is sound.*

Theorem 9 follows trivially from the following two lemmas, which are proven in Appendix D, and Theorem 5:

Lemma 6 *If all propagation rules in a backward Rhodium optimization O are sound then F_O as defined in* (6.2) *is sound.*

Lemma 7 *If all transformation rules in a backward Rhodium program O are sound, then R_O as defined in* (6.4) *is sound.*

As in the forward case, if all propagation rules of an optimization are sound, the induced analysis $\mathcal{A}_O$ is guaranteed to be sound, as stated in the following theorem.

Theorem 10 (backward analysis soundness) *If all propagation rules in a backward Rhodium optimization O are sound, then the analysis $\mathcal{A}_O$ is sound.*

The above four theorems and lemmas (Theorems 9 and 10, and Lemmas 6 and 7) all appear in Figure 4.1, where they are all labeled "Backward".

6.3.4 Implementation

The backward soundness conditions are encoded for the theorem prover in the same way as the forward soundness conditions. The only additional predicate that needs to be implemented is *someMeaningHolds*, and this is done by expanding the existential inside *someMeaningHolds* into a disjunction of all the declared backward-relational-fact schemas.

Chapter 7

EXECUTING RHODIUM OPTIMIZATIONS

Rhodium analyses and transformations are meant to be directly executable; they should not have to be reimplemented in a different language to be run. This chapter covers several aspects of Rhodium execution. Section 7.1 presents a simple flow-sensitive intraprocedural execution engine for Rhodium analyses and transformations. Sections 7.2 and 7.3 then show how Rhodium's flow-function model can be used to automatically generate provably sound flow-insensitive and/or interprocedural analyses.

The material presented in this chapter is not part of the main contribution of this dissertation – the main contribution is the Rhodium language design and the Rhodium soundness checker. The execution engine presented in Section 7.1 is a prototype implementation and lacks support for certain Rhodium features, for example user-defined merges. Furthermore, the ideas in Sections 7.2 and 7.3 have been adapted to the Rhodium system from previously known techniques, and they are at the design stage only – they have not been implemented yet. Nevertheless, this chapter gives an overview of what kinds of techniques can be used to execute Rhodium optimizations.

7.1 Intraprocedural flow-sensitive execution engine

Rhodium analyses and transformations are meant to be directly executable; they do not have to be reimplemented in a different language to be run. Using Whirlwind's framework for composable optimizations [63], I have implemented a forward intraprocedural execution engine for the core of the Rhodium language. Rhodium optimizations in Whirlwind peacefully co-exist with handwritten optimizations. By supporting such incremental adoption, it is possible to provide benefits to compiler writers even if the whole optimizer is not written in Rhodium.

The Rhodium execution engine interprets Whirlwind's intermediate representation as

Rhodium IR statements. In particular, the Whirlwind compiler represents a program using a control flow graph (CFG) and a dataflow graph (DFG). The execution engine treats each node n in Whirlwind's DFG as an Rhodium IL statement that assigns a value to the output edge of n based on input values to n. For example, a "plus" node in the DFG that has two incoming edges e_1 and e_2, and one outgoing edge e_3, is interpreted as a Rhodium IL assignment $e_3 := e_1 + e_2$. The nodes of the DFG are also threaded through the CFG, which is the graph that is used for iterative dataflow analysis. This way of viewing a compiler's intermediate representation through the lens of the Rhodium IL allows Rhodium analyses and optimizations to be executed on a variety of intermediate representations.

During iterative analysis, the Rhodium execution engine stores at each edge in the CFG an element of D (each element of D is a set of facts), and propagates facts across statements by interpreting the Rhodium rules. The engine's flow function $F_{exec} : \mathit{Node} \times D^* \rightarrow D^*$ takes a node n, and a tuple ds of incoming dataflow sets (one set for each incoming edge), and returns a tuple of outgoing dataflow sets (one set for each outgoing edge). The F_{exec} function is the implementation of the F function given in Equation (5.3) of Section 5.3.2. The engine's flow function operates as follows (where R_h is the set of forward rules that propagate on the outgoing edge $\mathit{out}[h]$):

$$F_{exec}(n, ds)[h] = \bigcup_{r \in R_h} \mathit{apply\_rule}(r, n, ds)$$

$$\mathit{apply\_rule}(\textbf{if } \psi \textbf{ then } f(t_1, \ldots, t_i)@\mathit{out}[h], n, ds) =$$
$$\textbf{let } \Theta = \mathit{sat}(\psi, n, ds, [\,]) \textbf{ in } \bigcup_{\theta \in \Theta} \{f(\theta(t_1), \ldots, \theta(t_i))\}$$

The flow function applies each rule separately and returns the union of the individual results. The *apply_rule* function computes all the facts propagated by a given rule. To do this, *apply_rule* first uses the *sat* function to compute all the satisfying substitutions that make the antecedent ψ hold. For each returned substitution θ, *apply_rule* adds the propagated fact, $f(\theta(t_1), \ldots, \theta(t_i))$, to the result set.

The $\mathit{sat} : \mathit{Pred} \times \mathit{Node} \times D^* \times \mathit{Subst} \rightarrow 2^{\mathit{Subst}}$ function (where we denote by *Pred* the set of all Rhodium predicates, and by *Subst* the set of all substitutions) finds satisfying substitutions: given a predicate ψ, a node n, a tuple ds of incoming sets of facts, and a

substitution θ, $sat(\psi, n, ds, \theta)$ returns the set of all substitutions θ' that have the following properties: (1) θ' makes ψ hold at node n when ds flows into n, or more formally, $[\![\psi]\!](n, ds, \theta')$ holds (2) θ' is an extension of θ and (3) the additional mappings in θ' are only for free variables of ψ. The original call to *sat* passes the empty substitution $[\,]$ for θ, and in this case $sat(\psi, n, ds, [\,])$ computes the set of all substitutions over the free free variables of ψ that make ψ hold at node n. Here are some representative cases from the implementation of *sat*:

$$
\begin{aligned}
sat(true, n, ds, \theta) &= \{\theta\} \\
sat(false, n, ds, \theta) &= \emptyset \\
sat(\psi_1 \vee \psi_2, n, ds, \theta) &= sat(\psi_1, n, ds, \theta) \cup sat(\psi_2, n, ds, \theta) \\
sat(\psi_1 \wedge \psi_2, n, ds, \theta) &= \textbf{let } \Theta = sat(\psi_1, n, ds, \theta) \\
&\qquad \textbf{in } \bigcup_{\theta' \in \Theta} sat(\psi_2, n, ds, \theta') \\
sat(t_1 = t_2, n, ds, \theta) &= unify(n, t_1, t_2, \theta) \\
sat(g(t_1, \ldots, t_j)@in[k], n, ds, \theta) &= \bigcup_{g(s_1,\ldots,s_j) \in ds[k]} unify\_terms(n, (t_1, \ldots, t_j), (s_1, \ldots, s_j), \theta) \\
sat(\exists x.\psi, n, ds, \theta) &= sat(\psi, n, ds, \theta \setminus x)[x \mapsto \theta(x)]
\end{aligned}
$$

The *sat* function makes use of a unification routine: the call $unify(n, t_1, t_2, \theta)$ attempts to unify $\theta(t_1)$ and $\theta(t_2)$. If the unification fails, then *unify* returns the empty set. If the unification succeeds with substitution θ', then θ' is augmented with all the mappings from θ to produce θ'', and *unify* returns the singleton set $\{\theta''\}$. The *unify_terms* function works like *unify*, except that it unifies a sequence of terms with another sequence. The unification procedure also uses the parameter n that is passed to *unify* and *unify_terms* to substitute the terms *currStmt* and *currNode* before doing the unification. Furthermore, *unify* tries to evaluate terms such as $applyBinaryOp(*, C_2, C_3)$ from Figure 2.8. If such a term can be evaluated, *unify* replaces the term with what it evaluates to, and then proceeds as usual. If such a term cannot be evaluated (because for example either C_2 or C_3 is not bound yet), then unification fails.

Universal quantifiers are handled by expanding them into conjunctions over the domain of the quantifier. This expansion is possible because the domain of quantified variables is

finite for any particular intermediate-language program. For existential quantifiers, the *sat* function locally skolemizes the quantified variable, and then proceeds with the body of the quantifier. Any mapping of the quantified variable introduced for satisfying the body of the quantifier is discarded in the resulting substitutions. In particular, I use $\theta \setminus x$ to denote θ with any mapping of x removed. I also use $\Theta[x \mapsto \theta(x)]$ to denote $\cup_{\theta' \in \Theta}\{\theta'[x \mapsto \theta(x)]\}$, where $\theta'[x \mapsto \theta(x)]$ stands for the substitution θ' updated so that it maps x in the same way that θ does: if θ maps x to a value, then $\theta'[x \mapsto \theta(x)]$ maps x to the same value, and if θ does not have a mapping for x, then neither does $\theta'[x \mapsto \theta(x)]$. Using this notation, the result of $sat(\exists x.\psi, n, ds, \theta)$ is therefore $sat(\psi, n, ds, \theta \setminus x)[x \mapsto \theta(x)]$.

The Rhodium execution engine is currently a prototype, and it is not complete. For example, it does not properly handle user-defined merges, backward analyses, backward optimizations, or rules involving the `currNode` term (as opposed to `currStmt`). Furthermore, I have not implemented any of the profitability heuristics mentioned in this dissertation. Finally, because the execution engine interprets rules, instead of generating specialized code to run the rules, it is too slow for practical use. There are several directions of future work, outlined in Section 10.3, for making the execution engine more efficient.

7.2 Flow-insensitive analysis

As mentioned in Section 2.1, propagation rules in a Rhodium optimization implicitly define a flow function. One of the benefits of such a flow-function-based model is that flow functions are a standard way of expressing analyses. Not only are compiler writers already familiar with flow functions, but there is also a wide variety of known implementation and theoretical techniques that apply to flow-function-based analyses. This section and the next show how two of these previously known techniques can be used in the context of Rhodium to generate sound flow-insensitive and interprocedural analyses.

The first benefit that falls out from Rhodium's flow-function model is that Rhodium can easily support provably sound flow-insensitive analyses, using a standard technique. In particular, a flow-sensitive flow function can always be run in a flow-insensitive manner. Instead of keeping a separate set of dataflow facts at each edge, the execution engine keeps a single set I for the whole procedure. Iterative analysis proceeds as usual, except that each

time a flow function is run, it takes I as input, and its result is merged into I. In this way one can produce a sound flow-insensitive analysis from a sound flow-sensitive version.

As a simple example, Figure 7.1 shows an IL code snippet, on which we will run the pointer analysis rules from Figure 7.2. To make the example easier to follow, the IL code and the propagation rules use the statement form $*A := \&B$, which in the real IL would have to be decomposed into $T := \&B; *A := T$.

The left side of Figure 7.3 shows the results of the pointer analysis running in a flow-sensitive mode. Each program point between two statements is annotated with a set of dataflow facts. The left-most column shows the contents of these sets explicitly, where the *mustNotPointTo* tags have been left out for brevity. Thus, the information coming into `a` := &`b` is the full set of *mustNotPointTo* facts, whereas the information after `a` := &`b` contains all *mustNotPointTo* facts, except for *mustNotPointTo*(`a`, `b`). The numbered boxes indicate which facts were generated by which rules. Tuples in a box numbered n were generated by the n^{th} rule of Figure 7.2. Next to the explicit set representation, Figure 7.3 also shows the results of the pointer analysis as may-points-to graphs. These graphs represent the complement of the *mustNotPointTo* sets.

The right side of the figure shows the first iteration of running the pointer analysis in flow-insensitive mode, where the execution engine only keeps track of one global set of facts. The diagram displays this global set after each statement has been analyzed. The main differences occurs after analyzing `a` := &`c`: in the flow-sensitive version, `a` does not point to `b`, whereas in the flow-insensitive version, the *mustNotPointTo*(`a`, `b`) fact is lost because the resulting set is merged (intersected) into the global set. Because the global set has changed while analyzing statements during the first iteration, the execution engine must analyze the statements again. During the second iteration, which is not shown in the diagram, the global set does not change, and so the final result of the flow-insensitive analysis is the set from the bottom right corner of the diagram. This result is imprecise, in that it smears information from one program point to another. This loss in precision, however, is counter-balanced by gains in efficiency. Flow-insensitive analyses can run a lot faster than their flow-sensitive counterparts, mostly because of reduced memory usage.

```
a := &b;
*a := &b;
a := &c;
*a := &b;
```

Figure 7.1: Sample IL code snippet

define forward edge fact
mustNotPointTo(X:*Var*, Y:*Var*)
with meaning $\eta(X) \neq \eta(\&Y)$

① **if** $stmt(X := \&Z) \wedge Y \neq Z$
then $mustNotPointTo(X,Y)@out$

② **if** $mustNotPointTo(X,Y)@in \wedge mustNotDef(X)$
then $mustNotPointTo(X,Y)@out$

③ **if** $stmt(*A := \&B) \wedge B \neq Y \wedge$
$mustNotPointTo(X,Y)@in \wedge$
then $mustNotPointTo(X,Y)@out$

Figure 7.2: Pointer analysis rules for the code in Figure 7.1

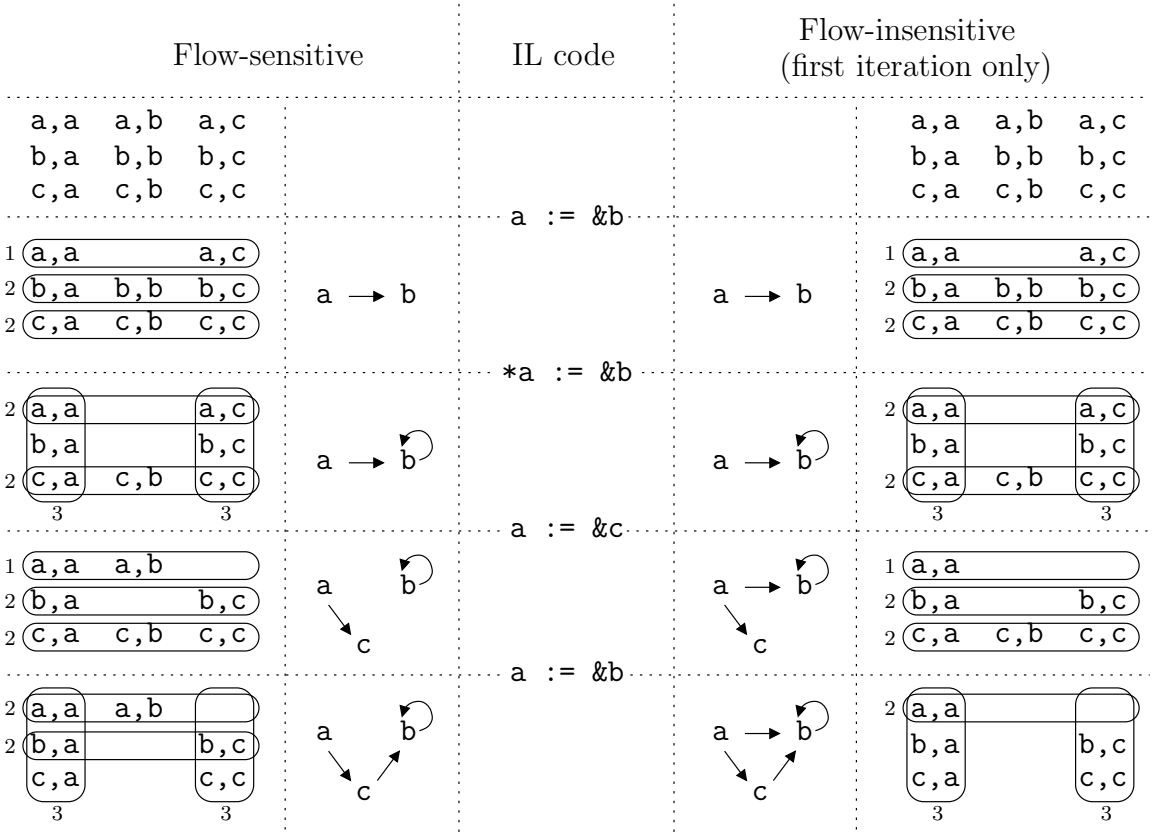


Figure 7.3: Results of flow-sensitive and flow-insensitive pointer analysis

7.3 Interprocedural analysis

Rhodium's flow-function model also makes it possible to adapt a previous flow-function-based framework [23] from the Vortex compiler [33] in order to automatically build provably sound interprocedural analyses in Rhodium. The Vortex framework is capable of automatically creating an interprocedural analysis from the intraprocedural version and a context-sensitivity strategy. The Vortex framework has been used to write realistic interprocedural analyses, such as various kinds of class analyses [47], constant propagation, side-effect analysis, escape analysis, and various synchronization-related analyses [4].

The key insight here is that this framework can be proven sound by hand once and for all, for a set of predefined context-sensitivity strategies. As a result, any interprocedural analysis generated by one of these predefined strategies is guaranteed to be sound provided the intraprocedural version is. To build a provably sound interprocedural analysis, the programmer therefore writes the intraprocedural version in Rhodium, making sure that it passes all the soundness checks, and then picks one of the predefined context-sensitivity strategies. The interprocedural framework would then automatically generate an interprocedural version of the analysis that is guaranteed to be sound.

The interprocedural framework is parameterized by a context-sensitivity strategy that describes what context a function should be analyzed in at a particular call site. The context-sensitivity strategy is embodied in a function *selectCalleeContext*. Given a call site n, the context $c \in \mathit{Context}$ in which the caller is being analyzed, and the dataflow information d at the call site, $\mathit{selectCalleeContext}(n, c, d)$ returns the context for analyzing the callee at this call site.

Table 7.1 shows the definition of *Context* and *selectCalleeContext* for two commonly used context-sensitivity strategies: the transfer function strategy (also known as Sharir and Pnueli's functional approach [102]), and Shivers's k-CFA algorithm [103] (also known as the k-deep call-strings strategy of Sharir and Pnueli [102]). The context-insensitive strategy can be achieved using 0-CFA.

The Rhodium interprocedural framework operates by creating an interprocedural flow function F_i from an intraprocedural version F. Instead of propagating facts $d \in D$, the

Table 7.1: Definition of *Context* and *selectCalleeContext* for two common context-sensitivity strategies.

Strategy	*Context*	*selectCalleeContext*
Transfer function	D	$selectCalleeContext(n, c, d) = d$
k-CFA	$list[string]$	$selectCalleeContext(n, c, d) = last(cat(c, [fnOf(n)]), k)$ where: $cat(l_1, l_2)$ concatenates lists l_1 and l_2, $fnOf(n)$ returns the name of the enclosing function containing n, $last(l, k)$ returns the sublist containing the last k elements of l (or l if l contains fewer than k elements)

interprocedural analysis propagates functions $cd \in Context \rightharpoonup D$ which map a calling context c to the dataflow information d that holds in that context. The elements of D form a lattice, whose ordering operator we denote by $\sqsubseteq$. Although in theory, the algorithm propagates functions, in practice, the algorithm represents these functions using partial maps. The partial map $[c_1 \mapsto d_1, c_2 \mapsto d_2, \ldots, c_i \mapsto d_i]$ represents the function:

$$
\begin{array}{l}
\lambda(c)\ .\ \textbf{if}\ (c \sqsubseteq c_1)\ d_1 \\
\qquad\quad \textbf{elsif}\ (c \sqsubseteq c_2)\ d_2 \\
\qquad\quad \ldots \\
\qquad\quad \textbf{elseif}\ (c \sqsubseteq c_n)\ d_n \\
\qquad\quad \textbf{else}\ \top
\end{array}
$$

For nodes that are not function calls or returns, F_i simply evaluates F pointwise on each range d element. For a call node n, for each $(c \mapsto d)$ pair flowing into the call, F_i merges (pointwise) the pair $(selectCalleeContext(n, c, d) \mapsto d)$ into the map on the entry edge of the callee's CFG, which will cause the callee to be further analyzed if the edge information changes. For a return node, for each $(c' \mapsto d')$ pair flowing into the return, for each call site n and inflowing pair $(c \mapsto d)$ such that $c' = selectCalleeContext(n, c, d)$, the pair $(c \mapsto d')$ is merged into the map on n's successor edge.

Figure 7.4 shows a simple example of how the framework operates runs an interprocedu-

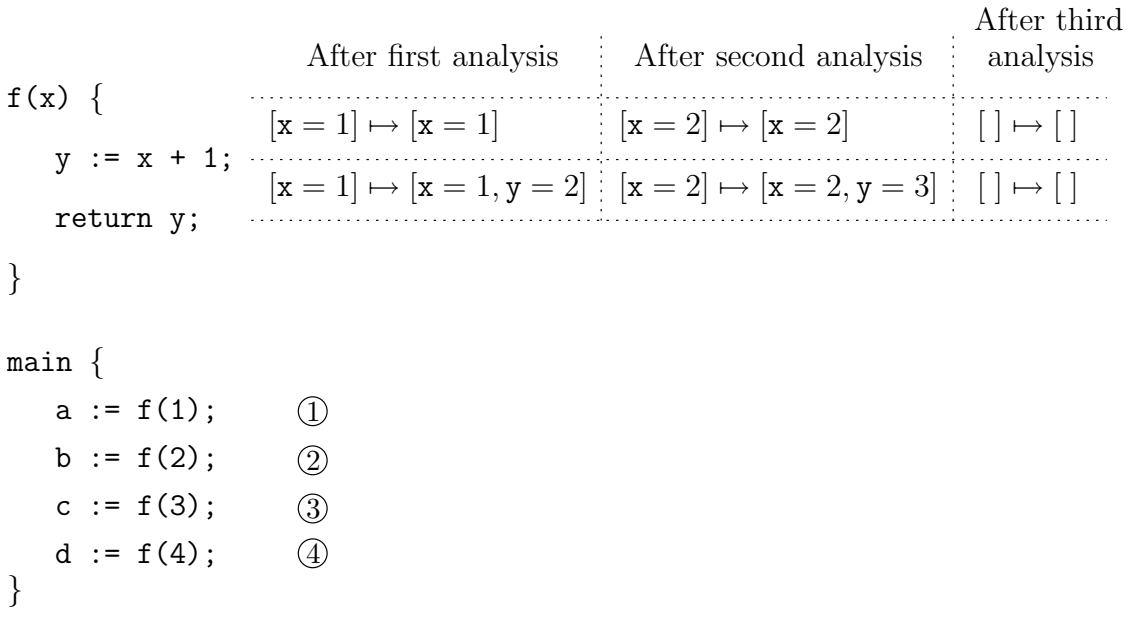


Figure 7.4: Results of flow-sensitive and flow-insensitive pointer analysis

ral constant-propagation analysis, using the transfer function strategy from Figure 7.1. The intraprocedural analysis that the framework starts with is a constant-propagation analysis written using the *hasConstValue* fact schema from Figure 2.3. For the purposes of this example, the set $\{hasConstValue(x_1, c_1), \ldots, hasConstValue(x_i, c_i)\}$ is denoted by $[x_1 = c_1, \ldots, x_i = c_i]$. Also, each horizontal line in the figure represents the final set computed by the analysis at the corresponding program point, with each column representing the additional information added by a re-analysis of `f`. For example, at the entry of `f`, the final result computed by the analysis will be $[\ [\mathtt{x} = 1] \mapsto [\mathtt{x} = 1], [\mathtt{x} = 2] \mapsto [\mathtt{x} = 2], [\ ] \mapsto [\ ]\ ]$. As will be explained shortly, this example uses a widening operator that limits to 2 the number of distinct contexts that `f` is analyzed in.

When the framework analyses the first call to `f`, on line 1, `f` has not been analyzed yet, so the analysis proceeds in the $[\mathtt{x} = 1]$ context, which results in `a` being 2 at the return site. For the second call to `f`, on line 2, the function has already been analyzed in the context $[\mathtt{x} = 1]$, but the new context, $[\mathtt{x} = 2]$, causes re-analysis of `f`, because the information at the entry of `f` has changed. In this case, the returned dataflow information will map `b` to 3 in the

caller. The pattern is now clear: every time `f` is called in a different dataflow information context, it will be re-analyzed. This context sensitivity strategy can be expensive, and in fact, may not even terminate. For example, if `f` contains a recursive call `f(x + 1)`, then `f` would be analyzed for successively increasing values of `x`. Analogously to widening operators as discussed in Section 2.5.3, one can enrich Rhodium by allowing optimization writers to specify a context widening operator to control the amount of context-sensitivity. For example, after k different contexts have been selected for a function, all future contexts could be widened to the most general context, [ ], bounding the number of times the function is analyzed. In the above example, if we assume that $k = 2$, then the third call to `f`, on line 3, will cause one final analysis of `f`, in the context [ ]. The fourth call to `f`, on line 4, would not cause any re-analysis, and the returned dataflow information at the call-site would be [ ].

Chapter 8

EVALUATION

In this chapter, I evaluate the Rhodium system along three dimensions: expressiveness, debugging value, and reduced trusted computing base.

8.1 Expressiveness

One of the key choices in the Rhodium system is to restrict the language in which optimizations can be written, in order to gain automatic reasoning about soundness. Unfortunately, a restricted language is inevitably less expressive than a general purpose language, and if the restrictions in the language are too onerous, they may reduce expressiveness to a point where the language is not useful anymore. To avoid this pitfall, I have carefully designed Rhodium so that it provides compiler writers with expressive power, despite its restricted nature. There are several aspects of the Rhodium system that enable this expressive power, while retaining automated soundness checking. First, much of the complexity of an optimization can be factored into the profitability heuristic, which is unrestricted. Second, Rhodium propagation rules provide a powerful way of expressing regular dataflow functions, which are commonly used by compiler writers to express many optimizations. Third, optimizations that traditionally are expressed as having simultaneous effects at multiple points in the program, such as various sorts of code motion, can in fact be decomposed into several simpler transformations, each of which fits Rhodium's model. The loop-induction-variable strength reduction from Section 2.3.1 illustrates all three of these points.

Even so, the current version of Rhodium does have limitations. For example, it cannot express some forms of many-to-many transformations, for example loop unrolling, loop tiling, and loop fusion. Also, optimizations that require more than first-order logic for the meaning cannot be expressed in Rhodium. For example, the meanings of some facts in shape analysis [52, 53, 24, 111, 97] require first-order logic plus transitive closure, which is

strictly more expressive than first-order logic. As a result, although Rhodium can express some simple kinds of heap summaries, it cannot express the more complicated kinds of heap summaries arising in shape-analysis.

Furthermore, optimizations and analyses that build complex data structures to represent their dataflow facts may be difficult to express. Rhodium programmers must build their data structures out of sets of tagged tuples. These simple sets are deceivingly expressive, being able to encode such complicated data structures as points-to graphs. Nevertheless, tagged tuples do have limitations. For example, they are not first-class values, and so they cannot be arguments to other tagged tuples. This restriction makes it difficult to express data structures that have user-defined nested structures.

Finally, it is possible for limitations in either the Rhodium proof strategy or in the automatic theorem prover to cause a sound optimization expressible in Rhodium to be rejected. For example, the Rhodium proof strategy for forward optimizations requires the forward concrete semantics to be preserved at each edge in the CFG of each procedure. As a result, forward optimizations that preserve the input-output behavior of a procedure, but not the semantics at internal edges of the procedure's CFG, cannot proven sound in the Rhodium system, even if they are expressible in Rhodium. Despite not being proven sound, such optimizations can nevertheless be run by the Rhodium execution engine.

In all of the above cases, optimizations can still be written outside of the Rhodium framework, perhaps verified using translation validation. Optimizations written in Rhodium and proven sound can peacefully co-exist with optimizations written "the normal way". The Whirlwind compiler has in fact many optimizations written outside of the Rhodium system.

8.2 Debugging benefit

Writing sound optimizations is difficult because there are many corner cases to consider, and it is easy to miss one. The Rhodium system in fact found several subtle problems in previous versions of certain optimizations. For example, consider the following $exprIsAvailable(X, E)$ fact schema, which captures the fact that an expression E is available in variable X:

define forward edge fact $exprIsAvailable(X{:}Var,\ E{:}Expr)$

with meaning $\eta(X) = \eta(E)$

decl X:*Var*, E:*Expr*

if $currStmt = [X := E]$
then $exprIsAvailable(X, E)@out$

The above rule, which at first sight may seem correct, in fact is unsound. The problem is that on a statement `x := x + 1`, the rule will propagate that `x` is equal to `x + 1`, which is obviously wrong. I wrote the above rule by mistake when implementing common subexpression elimination, and the soundness checker caught the bug. Other users of the Rhodium system have made similar corner-case mistakes. Although in retrospect the above rule is obviously wrong, such mistakes happen surprisingly often when writing many rules.

Furthermore, some of these mistakes can go undetected for a long time, because the particular corner case that triggers the bug may not occur in many programs. To illustrate such an example, consider a scenario where the Rhodium IL had expressions with nested derefences, for example `*x + *y`. The correct version of the above rule would be:

if $currStmt = [X := E] \wedge unchanged(E)$
then $exprIsAvailable(X, E)@out$

The $unchanged(E)$ node fact here makes sure that E is not modified by the statement $X := E$, and it uses pointer information in the case of expressions that have dereferences in them. Now, suppose that the programmer had instead written the following buggy rule:

if $currStmt = [X := E] \wedge varsInExprNotModified(E)$
then $exprIsAvailable(X, E)@out$

In this rule, the node fact $varsInExprNotModified(E)$ makes sure that the variables appearing syntactically in E are not modified. This rule is buggy because for a statement `z := *x + *y`, it would propagate the fact $exprIsAvailable(\texttt{z}, *\texttt{x} + *\texttt{y})$, which would be incorrect if `x` pointed to `z` (and `*y` was different from 0). However, it takes a *nested* dereference to uncover this bug — statements with top-level dereferences, like $X := *Y$, do not uncover the bug because if Y points to X, then the statement is a self-assignment, after which X and $*Y$ will be equal. As a result, if programs that use nested dereferences are rare, say

because the front-end of the compiler generates RISC-like statements [86], then the above bug could remain uncovered for a long time.

8.3 Reduced trusted computing base

In computer-security terminology, the trusted computing base (TCB) of a system is the part of the system that enforces the security policy. To compromise the security policy, an attacker must compromise some part of the trusted computing base. In the context of software validation, the TCB is that part of a system whose correctness guarantees that the system as a whole satisfies the property it is meant to satisfy. As a result, a bug outside the TCB cannot invalidate the property being ensured, whereas a bug inside the TCB may do so. If the property we are trying to ensure is the soundness of a compiler, then a traditional testing approach that provides no guarantees results in the entire compiler being part of the TCB. The Rhodium system moves the optimization phase, one of the most intricate and error-prone portions of the compiler, outside of the TCB. Instead, the trust in this phase has been shifted to three components: the soundness checker, including the automatic theorem prover, the manual proofs done as part of the Rhodium system, and the engine that executes optimizations. Because all these components are optimization-independent, new optimizations can be incorporated into the compiler without enlarging the TCB. Furthermore, as discussed in Section 7.1, the execution engine is implemented as a single dataflow analysis common to all user-defined optimizations. This means that the trustworthiness of the execution engine is akin to the trustworthiness of a single optimization pass in a traditional compiler.

Trust can be further enhanced in several ways. First, one could use an automatic theorem prover that generates proofs, such as the prover in the Touchstone compiler [79]. This would allow trust to be shifted from the theorem prover to a simpler proof checker. The manual proofs of the Rhodium system are made public for peer review in Appendices B, C, and D to increase confidence. One could also use an interactive theorem prover such as PVS [84] to validate these proofs.

Chapter 9

RELATED WORK

9.1 Correctness of program analyses and transformations

9.1.1 Manual techniques

A significant amount of work has been done on manually proving dataflow analyses and transformations sound, including abstract interpretation [29, 30, 31], the work on the VLISP compiler for PreScheme [48, 83], Kleene algebra with tests [60], manual proofs of soundness based on partial equivalence relations [15], and manual proofs of soundness for optimizations expressed in temporal logic [108, 109, 98, 99, 61].

In fact, the Rhodium line of research (including Rhodium's predecessor, Cobalt [64]) was inspired by recent work in this area by Lacey et al. [61]. Lacey describes a language for writing optimizations as guarded rewrite rules evaluated over a labeled CFG, and presents a general strategy, based on relating execution traces of the original and transformed programs, for manually proving the soundness of optimizations written in his language. Three example optimizations are shown and proven sound by hand using this strategy.

Unfortunately, the generality of Lacey's language and the associated proof strategy makes it difficult to automate. Lacey's guards may be arbitrary Computational Tree Logic (CTL) [25] formulas interpreted over the entire CFG. In contrast, antecedents of Rhodium rules can only refer to incoming and outgoing edge facts. The local nature of rules, combined with the semantic meaning of facts, makes Rhodium rules more amenable to automated soundness checking than Lacey's CTL guards. Finally, Lacey's language does not have the concept of profitability analyses. As a result, it may be possible that certain optimizations are expressible in Rhodium using profitability analyses, but not in Lacey's language because the profitability analyses are not expressible in CTL.

9.1.2 Semi-automated techniques

Analyses and transformations have also been proven correct mechanically, but not automatically: the correctness proof is performed with an interactive theorem prover that requires guidance from the user. For example, Young [124] has proven a code generator correct using the Boyer-Moore theorem prover enhanced with an interactive interface [56]. As another example, Cachera et al. [21] show how to specify static analyses and prove them correct in constructive logic using the Coq proof assistant. Via the Curry-Howard isomorphism, an implementation of the static analysis algorithm can then be extracted from the proof of correctness. Aboul-Hosn and Kozen present KAT-ML [1], an interactive theorem prover for Kleene Algebra with Tests, which can be used to interactively prove properties of programs. In all these cases, however, the proof requires help from the user. In contrast, Rhodium's proof strategy is fully automated.

9.1.3 Fully automated techniques

Previous automated techniques for guaranteeing compiler soundness, such as translation validation [91, 77, 126, 125, 45] and credible compilation [95, 94], have focused on checking the soundness of a given compilation run, rather than checking that the compiler is always sound. In credible compilation, the compiler produces a proof that the optimized output program has the same meaning as the original input program. This proof can then be checked using a proof checker. In translation validation, a validator module observes the intermediate-language programs before and after each transformation, and tries to show that all these programs have the same meaning. Translation validation is a less intrusive technique than credible compilation, since the validator module requires little or no help from the compiler: the validator only needs to know the intermediate-language program before and after each transformation, and compilers often have command-line switches to produce this information.

Because translation validation and credible compilation both check soundness one compilation at a time, a bug in the optimizer only appears when the compiler is run on a program that triggers the bug. The Rhodium system allows optimizations to be proven

sound before the compiler is even run once. However, to do so, optimizations must be written in a special-purpose language. Furthermore, the Rhodium execution engine becomes part of the trusted computing base, while translation validation and credible compilation do not require trust in any part of the optimizer.

Rhodium's current expressiveness limitations are very similar to the limitations that earlier translation validation frameworks suffered from. For example, Necula's [77] translation validator for gcc could only prove the soundness of so-called structure-preserving transformations, meaning that the structure of loops and branches had to remain the same. Rhodium's local transformation model, where one statement is replaced with another, imposes essentially the same restriction. Advances in the state of the art have gradually lifted many of these early limitations of translation validators. For instance, the Tvoc translation validation infrastructure [45] shows how to perform translation validation of both structure-preserving transformations, as well as many kinds of loop optimizations, including loop fusion, loop interchange and loop tiling. As future work, I plan to investigate how these newer translation-validation techniques could be adapted to Rhodium in order to support non-structure-preserving transformations, such as loop optimizations.

Proof-carrying code [76], certified compilation [78], typed intermediate languages [114], and typed assembly languages [73, 72] have all been used to automatically prove properties of programs generated by a compiler. However, the kinds of properties that these approaches have typically guaranteed are type safety or memory safety. In our work, we prove the stronger property of semantic equivalence between the original and transformed programs.

9.2 Languages and frameworks for specifying analyses and transformations

The idea of analyzing optimizations written in a domain-specific language was introduced by Whitfield and Soffa with the Gospel language [121]. Their framework, which is called Genesis, can automatically examine the interactions between different optimizations written in Gospel, with the goal of determining the best order in which to run them. In particular, by analyzing the pre- and post-conditions of optimizations, their framework can determine if one optimization helps or hinders another optimization. This information can then be used to select an order for running optimizations that will maximize helpful interactions

and minimize hindering ones. The main differences between Rhodium and the Gospel work stem from the difference in focus: the Rhodium system explores techniques for checking the soundness of optimizations, whereas Whitfield and Soffa have explored techniques for analyzing optimization dependencies.

Attribute grammars [58, 35, 34] are another method for specifying analyses that is related to Rhodium. Attribute grammars provide a general mechanism for annotating nodes of a parse tree with information. An attribute grammar extends a regular grammar with semantic rules stating how attributes of terminals and non-terminals appearing in a production are related to each other. The attributes at each node in the parse tree are computed by solving the constraints that these semantic rules impose. Attribute grammars are a very general mechanism for computing information, and they have in fact been used to specify and solve dataflow analysis problems [9, 10, 22]. Rhodium propagation rules are very similar to semantic rules in attribute grammars. However, semantic rules in attribute grammars are not automatically connected to the semantics of the parse tree, as Rhodium propagation rules are.

Many other frameworks and languages have been proposed for specifying dataflow analyses and transformations, including Sharlit [115], SPARE [117], FIAT [49], McCAT [51], System-Z [123], PAG [5], the k-tuple dataflow analysis framework [67], Dwyer and Clarke's system [38], languages based on regular path queries [106], languages based on temporal logic [108, 61], and the language of Whaley and Lam [120] based on Prolog. Although all these approaches make it easier to write and reason about optimizations, thus reducing the potential for human errors, none of these approaches addresses automated soundness checking of the specified transformations.

9.3 Automated theorem proving and applications

There has been a long line of work on automated theorem proving techniques, dating back to the early 1950s. Some of the big areas of research in this work include handling quantifiers in first-order logic [36, 101]; applying induction in a fully automated way [57, 100, 75]; applying the resolution rule efficiently and effectively [96, 118]; dealing with equality and equational theories [36, 66, 59]; developing decision procedures for

various domains [80, 55, 112, 105, 122, 32]; handling the communication between different decision procedures [81, 104]; handling the communication between decision procedures and the heuristic prover [16, 41, 13]; reducing non-determinism in the proof systems, and therefore the searches through these proof systems [88, 37]; and determining heuristics for handling the remaining non-determinism [88, 37]. State-of-the art theorem provers today are large, sophisticated systems, many of which have been developed and improved over several decades. Some of these systems include PVS [85], Nuprl [26], Twelf [89, 100], Simplify [36], the Boyer Moore theorem prover [56, 57], Isabelle [87], HOL [46], Vampire [92], and Spass [119].

Automated theorem provers have also been put to use in many applications. They have been used to solve open problems in mathematics, such as Robbin's problem in boolean algebra [69], which was open since the 1930s, and various open problems about quasigroups [107]. They have also been used to prove interesting properties about real-world systems, properties that would have been hard, difficult or tedious to prove by hand. For example, automated theorem provers have been used to verify microprocessors [17, 18, 85], communication protocols [18], concurrent algorithms [18], and various properties of software systems [42, 12, 54, 85, 89]. This is the line of research that the Rhodium system belongs to.

The Rhodium soundness checker uses Simplify [36], the automatic theorem prover used in the Extended Static Checker for Java (ESC/Java) [42]. Simplify is a fully automated theorem prover for first-order logic based on the Nelson-Oppen architecture for combining decision procedures [81]. In the context of ESC/Java, Simplify is used to determine whether or not a given precondition holding before a sequence of statements implies a given postcondition after the statements. Simplify's heuristics for pruning the search space and for instantiating quantifiers have been heavily tuned for this purpose. Since Rhodium's proof obligations are of the same nature as the ones in ESC/Java (show that some postcondition follows from some precondition), Simplify has been very effective at discharging Rhodium's proof obligations.

Simplify has a few drawbacks, however. First, the input language to Simplify is untyped. Not only does this mean that typing information has to be formalized explicitly, leading to

longer and more complicated axioms, but it also means that misspelled variable names in the axioms don't get discovered until the theorem prover fails to prove an obligation. Second, Simplify does not generate proofs, and as a result it cannot be removed from the trusted computing base by simply checking its proofs. Generated proofs can also be useful to the developers of the Rhodium system, for making sure that the theorem prover is following the expected proof path. Finally, Simplify does not prevent inconsistencies in the background axioms. If the background axioms are inconsistent, then Simplify can prove anything. To gain confidence that this does not happen, the Rhodium system checks the sanity of the generated background axioms, by asking Simplify to prove false in the context of these axioms. If the theorem prover succeeds, then there is an inconsistency in the background axioms that needs to be debugged. Another consistency check used in the Rhodium system is to make sure that the soundness checker properly fails on a variety of rules that are known to be buggy.

There are other theorem provers that do not have these drawbacks, but they are not well-suited for the purposes of the Rhodium system. CVC [113] and its successor, CVC Lite [14], are fully automated theorem provers based on the Nelson-Oppen architecture, but they also have built-in support for types. However, their heuristics for handling quantifiers are not as well-tuned as Simplify's. Many theorem provers generate proofs, but most of these provers are interactive. Of the ones that generate proofs *and* are fully automatic, like CVC [113] and CVC Lite [14], none have heuristics as well-tuned for pre/postcondition obligations as Simplify.

Chapter 10

CONCLUSION

This dissertation has shown that it is possible to check the soundness of compiler optimizations automatically if these optimizations are written in a specialized language. The work presented here, however, is only a small step in a much broader research agenda aimed at making good compilers easier to develop. This concluding chapter provides a glimpse of some of the future opportunities for research in the Rhodium project.

10.1 Increasing expressiveness

There are many opportunities for increasing Rhodium's expressive power. One direction would be to add support for one-to-many transformations, such as inlining or peep-hole optimizations that translate a single statement into a small sequence of simpler statements. In the same vein, one could also add support for many-to-many transformations, such as loop unrolling, loop interchange and loop tiling. As previously mentioned in Section 9.1.3, it may be possible to adapt techniques from translation validation to prove the soundness of loop transformations in Rhodium. Another way of supporting loop optimizations would be to use a different intermediate representation, such as a program dependence graph [40], where these optimizations are expressed as a one-to-one transformation. For example, in a program dependence graph, loop unrolling can be expressed by replacing the region node for the loop with a new region node that has more children. The challenge in supporting such transformations is to devise a proof strategy for the new IR that will be amenable to automation.

Using different program representations can also improve expressiveness in other ways. If one had an AST representation or a dataflow graph representation, the programmer could write interesting analyses and transformations over these graphs, rather than simply over the CFG. In addition to supporting new kinds of IRs, the Rhodium language could also

support transformations from one IR to another, for example from ASTs to CFGs. If the refinement relation were defined appropriately, this would also allow the Rhodium system to support provably correct refinement-based programming [11, 71], in which an efficient executable is generated from a high-level specification by applying a sequence of refinements expressed in the Rhodium language. The Rhodium system could also provide automatic or semi-automatic support for converting an optimization that runs on one representation to an optimization running on another.

Another direction for improving the expressiveness of Rhodium would be to provide better mechanisms for building user-defined data structures. For example, one could make tagged tuples first-class, so that they can be used a parameters to other tagged tuples. The main challenge in doing so would be to develop a mechanism for expressing the meaning of a tagged "parent" tuple in terms of the meanings of its tagged "children" tuples.

10.2 Checking properties other than soundness

We all want compilers to be sound, but we also want them to produce good-quality code. The Rhodium system has so far focused on providing strong guarantees about the former, but not the latter. One broad direction of future work would be to provide static guarantees about the quality of the generated code. For example, if the programmer had some way of specifying what scenarios are important for performance, then the system may be able to check that under those scenarios, optimizations cannot degrade performance. As another example, the system may be able to quantify how precise propagation rules are, and flag rules whose precision could be improved.

In the same vein, there are other static properties of analyses or transformations that one may want to check. For example, soundness is not strictly necessary in bug-finding tools, and in such cases one may want to purposely give up soundness to gain scalability. However, doing so in a unprincipled way can lead to mistakingly giving up too much soundness, and therefore losing opportunities for finding bugs. This pitfall could be avoided by specifying formally the ways in which an analysis is intended to be unsound, and then checking statically that the analysis is unsound only in these expected ways.

10.3 Efficient execution engine

Another direction for future work is to explore more efficient implementation techniques for the Rhodium execution engine. For example, one could develop tools that perform offline partial evaluation of propagation rules with respect to the various statement kinds, and then generate an efficient flow function that dispatches based on the statement being analyzed to a specialized set of rules. One could also develop tools that make analyses scalable by examining their run-time traces and then using the gathered information to guide the automatic application of various representation optimizations (such as switching to BDDs [20] or bit-vectors for encoding Rhodium dataflow information). Finally, one could develop tools to explore automatically (or semi-automatically) the tradeoffs between the scalability and precision of a Rhodium analysis.

10.4 Inferring parts of the compiler

By providing static checks of soundness, The Rhodium system makes it easier to write sound program analyses and transformations. One could take this idea one step further by having the system automatically *generate* sound optimizations, rather than have the programmer write them in the first place. Despite a long line of work on generating various parts of a compiler from specifications [90, 62, 39, 2, 19, 44, 43], there has been little work on automatically *inferring* analyzers and optimizers, and on doing so in a way that guarantees their soundness. Some recent advances in this poorly explored direction are nevertheless encouraging, including the recent work on inferring compiler heuristics using genetic programming [27, 28, 110]. There are many more opportunities for novel research in this area.

In particular, colleagues and I are currently taking the first step in generating Rhodium optimizations automatically: instead of requiring programmers to *write* the rules for propagating facts across statements, a tool will *infer* these rules automatically from the fact schemas and their associated semantic meanings [93]. The algorithm for automatically generating Rhodium rules employs a search technique similar in nature to those used in automatic theorem provers. For the fact schemas we considered, our automatically generated

rules covered more than half the cases of the manually written rules, and automatically generated rules frequently included cases that were not covered by the handwritten rules.

To push this idea even further, one could try to infer fact schemas from a specification of the desirable transformations. And finally, one could infer these desirable transformations from a high-level goal-directed specification, or from examples showing the desired output of the optimizer on some sample input programs. The end result would be that an entire optimizer could be generated automatically from a high-level specification, all in a way that guarantees its soundness.

10.5 Extensible compilers

Another research direction would be to use the Rhodium system as a way of empowering end-user programmers with the ability to create their own domain-specific analyses and transformations. Oftentimes, it is the end programmer who has the application-domain knowledge and the coding-pattern knowledge necessary to implement useful analyses. This dissertation can provide a foundation for extensible compilers, and more broadly, extensible program analysis tools: end programmers, with little or no knowledge of program analysis, can now easily extend existing compilers or program analysis tools without fear of breaking them. One can use this foundation for investigating how domain-specific static checking and domain-specific optimizations can be made practical and useful.

BIBLIOGRAPHY

[1] Kamal Aboul-Hosn and Dexter Kozen. KAT-ML: An interactive theorem prover for Kleene algebra with tests. In *Proceedings of the 4th International Workshop on the Implementation of Logics (WIL'03)*, University of Manchester, September 2003.

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[4] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium*, pages 19–38, Venice Italy, September 1999.

[5] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50, Glasgow, Scotland, September 1995. Springer-Verlag.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Languge*. PhD thesis, DIKU, University of Copenhagen, May 1994 (available as DIKU technical report 94-19).

[7] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[8] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language (Fourth Edition)*. Addison-Wesley Professional, 2005.

[9] W. Babich and M. Jazayeri. The method of attributes for data flow analysis, part I: Exhaustive analysis. *Acta Informatica*, 10(3):245–264, October 1978.

[10] W. Babich and M. Jazayeri. The method of attributes for data flow analysis, part II: Demand analysis. *Acta Informatica*, 10(3):265–272, October 1978.

[11] Ralph-Johan Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.

[12] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[13] C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In *4th International Workshop on Frontiers of Combining Systems (FroCoS 02)*, volume 2309 of *Lecture Notes in Computer Science*, pages 132–146. Springer-Verlag, January 2002.

[14] Clark Barrett and Sergey Berezin. CVC lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, 2002.

[15] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice Itally, January 2004.

[16] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[17] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.

[18] R.S. Boyer and J.S. Moore. *A Computational Logic, Second Edition*. Academic Press, 1998.

[19] Doug Brown, John Levine, and Tony Mason. *lex & yacc*. O'Reilly, 1992.

[20] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[21] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Proceedings of the 13th European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[22] Martin D. Carroll and Barbara G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Proceedings of the 15th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego CA, January 1988.

[23] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report UW-CSE-96-11-02, University of Washington, November 1996.

[24] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[25] E. M. Clarke, E. A. Emerson, and A. P Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[26] R.L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[27] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Atlanta GA, May 1999.

[28] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, August 2002.

[29] 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 Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles CA, January 1977.

[30] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, January 1979.

[31] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, January 2002.

[32] D. Cyrluk, M.O. Moller, and H. Rue. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 97)*, volume 2754 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, June 1997.

[33] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose CA, October 1996.

[34] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Williamsburg VA, January 1981.

[35] Pierre Deransart and Martin Jourdan, editors. *Attribute Grammars and their Applications, International Conference WAGA, Paris, Fance, September 19-21, 1990, Proceedings*, volume 461 of *Lecture Notes in Computer Science*. Springer, 1990.

[36] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *Journal of the Association for Computing Machinery*, 52(3):365–473, May 2005.

[37] D. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, 1991.

[38] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *18th International Conference on Software Engineering*, pages 554–564, Berlin, Germany, March 1998.

[39] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG – a generator for efficient back ends. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, Portland OR, June 1989.

[40] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[41] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer-Verlag, January 2003.

[42] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[43] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[44] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[45] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, May 2005.

[46] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification Verification and Synthesis*, Kluwer International Series in Engineering and Computer Science, pages 73–128. Kluwer Academic Publishers, 1988.

[47] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 2001.

[48] J. Guttman, J. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Compucation*, 8(1-2):33–110, 1995.

[49] M.W. Hall, J.M. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A framework for interprocedural analysis and transformation. In *The Sixth Anunual Workshop on Parallel Languages and Compilers*, August 1993.

[50] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Professional, 2003.

[51] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A practical context-sensitive interprocedural analysis framework for C compilers. Technical Report ACAPS Technical Memo 72, McGill University School of Computer Science, July 1993.

[52] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 244–256, San Antonio TX, January 1979.

[53] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 66–74, Albuquerque, NM, January 1982.

[54] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[55] D. Kaplan. Some completeness results in the mathematical theory of computation. *Journal of the Association for Computing Machinery*, 15(1):124–134, January 1968.

[56] M. Kauffmann and R.S. Boyer. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, January 1995.

[57] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[58] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1971. Correction: *Mathematical Systems Theory*, 5(1):95–96, March 1971.

[59] D. Knuth and P.B. Bendix. Simple word problems in universal algebras. In *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[60] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Langauges and Systems*, 19(3):427–443, September 1997.

[61] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[62] Peter Lee. *Realistic Compiler Generation*. MIT Press, 1989.

[63] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, January 2002.

[64] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego CA, June 2003.

[65] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Long Beach CA, January 2005.

[66] Z. Manna and S.Ness. On the termination of markov algorithms. In *Proceedings of the International Conference on System Science*, pages 789–792, 1970.

[67] Stephen P. Masticola, Thomas J. Marlowe, and Barbara G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Langauges and Systems*, 17(5):777–803, September 1995.

[68] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In T. J. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, January 1967.

[69] W. W. McCune. Solutions of the Robbins problem. *Journal of Automated Reasoning*, 19(3):262–276, 1997.

[70] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston MA, January 1973.

[71] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[72] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta GA, May 1999.

[73] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[74] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[75] D. R. Musser. On proving inductive properties of abstract data types. In *Procecedings of the 7th ACM Symposium on Principles of Programming languages*, pages 154–162, Las Vegas NV, 1980.

[76] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[77] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 83–95, Vancouver, Canada, June 2000.

[78] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[79] George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In *Proceedings of the International Conference on Automated Deduction*, pages 25–44, Pittsburgh, Pennsylvania, June 2000. Springer-Verlag LNAI 1831.

[80] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[81] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[82] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[83] D. P. Oliva, J. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1-2):111–182, 1995.

[84] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[85] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In M. A. McRobbie and J.K. Slaney, editors, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[86] David A. Patterson and David Ditzel. The case for the reduced instruction set computer. *Computer Architecture News*, 8(6):25–33, October 1980.

[87] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecure Notes in Computer Science*. Springer Verlag, 1994.

[88] F. Pfenning. Course notes on Automated Theorem proving, `http://www.cs.cmu.edu/~fp/courses/atp`. 2004.

[89] F. Pfenning and C. Schurmann. Sytsem description: Twelf – A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer-Verlag, July 1999.

[90] Uwe F. Pleban and Peter Lee. An automatically generated, realistic compiler for an imperative programming language. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, Atlanta GA, June 1988.

[91] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, 1998.

[92] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI communications*, 15(2-3):91–110, 2002.

[93] Erika Rice, Sorin Lerner, and Craig Chambers. Automatically inferring sound dataflow functions from dataflow fact schemas. In *Informal Proceedings of the 4th International Workshop on Compiler Optimization meets Compiler Verificaiton (COCV '05)*, Edinburgh, Scotland, April 2005.

[94] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology, March 1999.

[95] Martin Rinard and Darko Marinov. Credible compilation. In *Proceedings of the FLoC Workshop Run-Time Result Verification*, July 1999.

[96] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[97] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[98] David A. Schmidt. Dataflow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego CA, January 1998.

[99] David A. Schmidt and Bernhard Steffen. Data flow analysis as model checking of abstract interpretations. In Giorgio Levi, editor, *Proceedings of the 5th International Symposium on Static Analysis (SAS)*, volume 1503 of *Lecture Notes in Computer Science (LNCS)*, pages 351–380. Springer-Verlag, September 1998.

[100] C. Schurmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner H. Kirchner, editor, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Computer Science*, pages 286–300. Springer-Verlag, July 1998.

[101] N. Shakar, S. Owre, J.M. Rushby, and W. D. J. Stringer-Calvert. PVS Prover Guide - Version 2.4, `http://pvs.csl.sri.com/manuals.html`. 2001.

[102] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[103] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta GA, June 1988.

[104] R. E. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, January 1984.

[105] G. Sierksma. *Linear and Integer Programming - Theory and Practice*. Monographs and textbooks in pure and applied mathemtics. Marcel Dekker, New York, NY, 1996.

[106] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice Italy, January 2004.

[107] J. K. Slaney, M. Fujita, and M. E. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Compuers and Mathematics with Applications*, 29(2):115–132, 1995.

[108] Bernhard Steffen. Data flow analysis as model checking. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Science (TACS), Sendai (Japan)*, volume 526 of *Lecture Notes in Computer Science (LNCS)*, pages 346–364. Springer-Verlag, September 1991.

[109] Bernhard Steffen. Generating dataflow analysis algorithms for model specifications. *Science of Computer Programming*, 21(2):115–139, 1993.

[110] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego CA, June 2003.

[111] Jan Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Information and Computation*, 101(1):70–102, November 1992.

[112] A. Stump, C. Barrett, D. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceeding of the 16th Symposium on Logic in Computer Science*, pages 29–37. IEEE Computer Society Press, 2001.

[113] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002.

[114] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, Philadelphia PA, May 1996.

[115] Steven W. K. Tjiang and John L. Hennessy. Sharlit – A tool for building optimizers. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 82–93, July 1992.

[116] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Volume I.* Computer Science Press, 1988.

[117] G. A. Venkatesh and Charles N. Fischer. SPARE: A development environment for program analysis algorithms. *IEEE Transactions on Software Engineering*, 18(4):304–318, April 1992.

[118] A. Voronkov. Implementing bottom-up procedures with code trees: a case study of forward subsumption. Technical Report 88, Uppsala University, 1994.

[119] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. Spass version 2.0. In M. A. McRobbie and J.K. Slaney, editors, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 275–279. Springer-Verlag, 2002.

[120] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Washington D.C., June 2004.

[121] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[122] L. Wolsey. *Integer Programming.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, NY, 1998.

[123] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

[124] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.

[125] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A translation validator for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 65(2):1–17, April 2002.

[126] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.

Appendix A

INPUT AND OUTPUT EDGE EXPANSION

This appendix gives the semantics of the *in* and *out* edge names in terms of indexed edges names. In particular, *in* and *out* are desugared by replacing every propagation rule r with *expand-edges*(r), where *expand-edges* is defined as follows:

expand-edges$(r) \triangleq$ **if** r is a forward rule **then** *exp-fwd*(r) **else** *exp-bwd*(r)

where:

$$\textit{exp-fwd}(\textbf{if } \psi \textbf{ then } \phi) \triangleq \bigcup_{S \in Stmts} \{\textbf{if } \textit{currStmt} = S \wedge \textit{exp-in}(\psi, S) \textbf{ then } \textit{exp-out}(\phi, S)\}$$

$$\textit{exp-bwd}(\textbf{if } \psi \textbf{ then } \phi) \triangleq \bigcup_{S \in Stmts} \{\textbf{if } \textit{currStmt} = S \wedge \textit{exp-out}(\psi, S) \textbf{ then } \textit{exp-in}(\phi, S)\}$$

$$\textit{exp-in}(\alpha, S) \triangleq \bigwedge_{0 \leq i < numInEdges(S)} \alpha[in \mapsto in[i]]$$

$$\textit{exp-out}(\alpha, S) \triangleq \bigwedge_{0 \leq i < numOutEdges(S)} \alpha[out \mapsto out[i]]$$

$$Stmts \triangleq \{\ \texttt{decl}\ X,\ \texttt{decl}\ X[Y],\ \texttt{skip},\ X := \texttt{new},\ X := \texttt{new array},$$
$$X := E,\ X := P(Y),\ \texttt{if}\ B\ \texttt{goto}\ L1\ \texttt{else}\ L2,\ \texttt{return}\ X\ \}$$

numInEdges$(S) \triangleq$ number of input edges of a statement S

numOutEdges$(S) \triangleq$ number of output edges of a statement S

The intuition behind *expand-edges* is that, given a propagation rule r, it generates a specialized version of r for each input-output edge pair of each statement type.

Appendix B

ADDITIONAL MATERIAL FOR THE ANALYSIS FRAMEWORK

B.1 Definitions

In this section, I give a definition for the solution function $S_{\mathcal{A}} : \mathit{Graph} \times D^* \to (\mathit{Edge} \to D)$, and the transformation function $T : RF_D \times \mathit{Graph} \times (\mathit{Edge} \to D) \to \mathit{Graph}$.

Definition 40 *Given an analysis $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$, a graph g and a tuple of dataflow values $\iota \in D^*$ for the input edges of g, $S_{\mathcal{A}}(g, \iota)$ is defined as follows.*

First, I define the interpretation function $Int : E_g \times (E_g \to D) \to D$ as in Cousot and Cousot [29]: given an edge e and the current dataflow solution m, Int computes the dataflow value for e at the next iteration. Int is defined as:

$$Int(e, m) = \begin{cases} \iota[k] & \text{if } \exists k.e = InEdges_g[k] \\ F(n, \overrightarrow{m}(in_g(n)))[k] & \text{where } e = out_g(n)[k] \end{cases}$$

The global flow function $FG : (E_g \to D) \to (E_g \to D)$ takes a map representing the current dataflow solution, and computes the dataflow solution at the next iteration. FG is defined as:

$$FG(m) = \lambda e.Int(e, m)$$

The global ascending flow function FGA is the same as FG, except that it joins the result of the next iteration with the current solution before returning. This ensures that the solution monotonically increases as iteration proceeds, even if F is not monotonic. FGA is defined as:

$$FGA(m) = FG(m) \mathbin{\widetilde{\sqcup}} m$$

Finally, the result of $S_{\mathcal{A}}$ is a fixed point of FGA (the least fixed point if F is monotonic):

$$S_{\mathcal{A}}(g, \iota) = \bigsqcup_{j=0}^{\infty} FGA^j(\widetilde{\bot})$$

where $\widetilde{\bot} \triangleq \lambda e.\bot$, $FGA^0 = \lambda x.x$ and $FGA^k = FGA \circ FGA^{k-1}$ for $k > 0$.

Definition 41 *Given a replacement function R, a graph g and some analysis results m, $T(R, g, m)$ is defined as follows. First, I introduce the update function $Update : Graph \times Node \times Graph \rightarrow Graph$, which is used to replace a single node in a graph. Given an original graph g, a node n and a replacement graph r for this node, $Update(g, n, r)$ returns the result of replacing the node n with r in old. Update is defined as follows:*

$$Update(g, n, r) = (N_{new}, E_{new}, in_{new}, out_{new}, \\ InEdges_{new}, OutEdges_{new})$$

where

$$N_{new} = (N_g \setminus \{n\}) \cup N_r$$
$$E_{new} = (E_g \cup E_r) \setminus (Elmts(InEdges_r) \cup Elmts(OutEdges_r))$$
$$\text{where } Elmts(tuple) = \{d \mid \exists i.tuple[i] = d\}$$
$$InEdges_{new} = InEdges_g$$
$$OutEdges_{new} = OutEdges_g$$
$$in_{new}(n') = \begin{cases} in_g(n') & \text{if } n' \in N_g - \{n\} \\ \overrightarrow{ReplIn}(in_r(n')) & \text{if } n' \in N_r \end{cases}$$
$$out_{new}(n') = \begin{cases} out_g(n') & \text{if } n' \in N_g - \{n\} \\ \overrightarrow{ReplOut}(out_r(n')) & \text{if } n' \in N_r \end{cases}$$

and

$$ReplIn(e) = \begin{cases} in_g(n)[k] & \text{if } \exists k.e = InEdges_r[k] \\ e & \text{otherwise} \end{cases}$$

$$ReplOut(e) = \begin{cases} out_g(n)[k] & \text{if } \exists k.e = OutEdges_r[k] \\ e & \text{otherwise} \end{cases}$$

I now define $Update_\epsilon$, a simple extension to Update that works correctly if the replacement graph is ϵ:

$$Update_\epsilon(g, n, r) = \begin{cases} Update(g, n, singleNodeGraph(n, stmtAt(n))) & \text{if } r = \epsilon \\ Update(g, n, r) & \text{otherwise} \end{cases}$$

Note that $Update_\epsilon$ creates a copy of the original node if the replacement graph is ϵ.

The graph returned by $T(R, g, m)$ is then simply the iterated application of $Update_\epsilon$ on all the nodes of g. Thus, $T(R, g, m)$ is defined by:

$$T(R, g, m) = IT(R, g, m, N_g)$$

where IT (which stands for IteratedT) is:

$$IT(R, g, m, N) = \begin{cases} IT(R, g_{new}, m, N - \{n\}) & \text{if } \exists n \in N \\ g & \text{if } N = \emptyset \end{cases}$$

with

$$g_{new} = Update_{\epsilon}(g, n, R(n, \overrightarrow{m}(in_g(n))))$$

B.2 Proofs

Theorem 4 *If the flow function of an analysis $\mathcal{A}$ is sound then $\mathcal{A}$ is sound.*

Proof

Let $\mathcal{A} = (D_a, \sqcup_a, \sqcap_a, \sqsubseteq_a, \top_a, \bot_a, \alpha, F_a)$ be an analysis where F_a is sound. Let $(g, \iota_c, \iota_a) \in Graph \times D_c^* \times D_a^*$ such that $\overrightarrow{\alpha}(\iota_c) \sqsubseteq_a \iota_a$. Also, suppose that FG_c and FG_a are the global flow functions in the definitions of $S_{\mathcal{C}}$ and $S_{\mathcal{A}}$ respectively (see Definition 40), and similarly for the global ascending flow functions FGA_a and FGA_c. We need to show that:

$$\widetilde{\alpha}\left(\bigsqcup_{j=0}^{\infty} FGA_c^j(\widetilde{\bot_c})\right) \sqsubseteq_a \bigsqcup_{j=0}^{\infty} FGA_a^j(\widetilde{\bot_a}) \tag{B.1}$$

Because α is join-monotonic, we have:

$$\widetilde{\alpha}\left(\bigsqcup_{j=0}^{\infty} FGA_c^j(\widetilde{\bot_c})\right) \sqsubseteq_a \bigsqcup_{j=0}^{\infty} \widetilde{\alpha}(FGA_c^j(\widetilde{\bot_c})) \tag{B.2}$$

Using (B.2) and transitivity of $\sqsubseteq_a$, to show (B.1), all we need to show is:

$$\bigsqcup_{j=0}^{\infty} \widetilde{\alpha}(FGA_c^j(\widetilde{\bot_c})) \sqsubseteq_a \bigsqcup_{j=0}^{\infty} FGA_a^j(\widetilde{\bot_a})$$

To do this, we show $\forall j \geq 0\,.\, \widetilde{\alpha}(FGA_c^j(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^j(\widetilde{\bot_a})$. We first establish a few facts.

- Since F_c is continuous, it is monotonic, and therefore $FG_c^j(\widetilde{\bot_c})$ is an ascending chain. Thus, we get:

$$\forall j \geq 0. FG_c^j(\widetilde{\bot_c}) = FGA_c^j(\widetilde{\bot_c}) \tag{B.3}$$

- Let $M_c = Edges_g \rightarrow D_c$, and $M_a = Edges_g \rightarrow D_a$. Since F_a is sound, it satisfies property (4.5), which combined with $\overrightarrow{\alpha}(\iota_c) \sqsubseteq_a \iota_a$ can be used to get:

$$\begin{array}{l} \forall (m_c, m_a) \in M_c \times M_a. \\ \quad \widetilde{\alpha}(m_c) \sqsubseteq_a m_a \Rightarrow \widetilde{\alpha}(FG_c(m_c)) \sqsubseteq_a FG_a(m_a) \end{array} \tag{B.4}$$

- Since $FGA_a(m) = FG_a(m) \sqcup_a m$, we have:

$$\forall m \in M_a . FG_a(m) \sqsubseteq_a FGA_a(m) \tag{B.5}$$

Now we can show $\forall i \geq 0 \,.\, \widetilde{\alpha}(FGA_c^j(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^j(\widetilde{\bot_a})$. We do this by induction on j.

- *Base case.* When $j = 0$, $FGA_c^0(\widetilde{\bot_c}) = \widetilde{\bot_c}$, and $FGA_a^0(\widetilde{\bot_a}) = \widetilde{\bot_a}$. Since $\widetilde{\alpha}(\widetilde{\bot_c}) = \widetilde{\bot_a}$ (because α is bottom-preserving), we then get $\widetilde{\alpha}(FGA_c^0(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^0(\widetilde{\bot_a})$

- *Inductive case.* Assume $\widetilde{\alpha}(FGA_c^j(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^j(\widetilde{\bot_a})$ for some $j \geq 0$. We need to show $\widetilde{\alpha}(FGA_c^{j+1}(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^{j+1}(\widetilde{\bot_a})$. The proof is as follows:

$$\begin{array}{lcll} & \widetilde{\alpha}(FGA_c^j(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^j(\widetilde{\bot_a}) & \\ \Leftrightarrow & \widetilde{\alpha}(FG_c^j(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^j(\widetilde{\bot_a}) & \text{using (B.3)} \\ \Leftrightarrow & \widetilde{\alpha}(FG_c(FG_c^j(\widetilde{\bot_c}))) \sqsubseteq_a FG_a(FGA_a^j(\widetilde{\bot_a})) & \text{using (B.4)} \\ \Leftrightarrow & \widetilde{\alpha}(FG_c(FG_c^j(\widetilde{\bot_c}))) \sqsubseteq_a FGA_a(FGA_a^j(\widetilde{\bot_a})) & \text{using (B.5)} \\ \Leftrightarrow & \widetilde{\alpha}(FG_c^{j+1}(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^{j+1}(\widetilde{\bot_a}) & \\ \Leftrightarrow & \widetilde{\alpha}(FGA_c^{j+1}(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^{j+1}(\widetilde{\bot_a}) & \text{using (B.3)} \end{array}$$

■

In the following proofs, it will be useful to consider $S_{\mathcal{C}}(g, \iota)$ as the computation of the least fixed point of a set of dataflow equations. In particular, I denote by $FG_{(g,\iota)}$ the global flow function FG from the definition of $S_{\mathcal{C}}(g, \iota)$ (Definition 40). Because F_c is monotonic, we have that $S_{\mathcal{C}}(g, \iota) = \bigsqcup_{j=0}^{\infty} FG_{(g,\iota)}^j(\widetilde{\bot})$.

A fixed point of $FG_{(g,\iota)}$ is a solution $X : E_g \to D_c$ to the equation $X = FG_{(g,\iota)}(X)$. Because X is map from edges to dataflow information, the equation $X = FG_{(g,\iota)}(X)$ is a representation of the traditional dataflow equations for the graph g with incoming information ι. A least fixed point m of $FG_{(g,\iota)}$ is a fixed point of $FG_{(g,\iota)}$ such that for all fixed points m' of $FG_{(g,\iota)}$, it is the case that $m \sqsubseteq_c m'$.

Because F_c is monotonic, the global flow function $FG_{(g,\iota)}$ is also monotonic, and as a result, $S_{\mathcal{C}}(g, \iota)$ computes the least fixed point of $FG_{(g,\iota)}$. This fact will become useful in the following proofs because it provides an alternate way of reasoning about $S_{\mathcal{C}}(g, \iota)$. In particular, if we can show that m is the least fixed point of $FG_{(g,\iota)}$, then it must be the case that $m = S_{\mathcal{C}}(g, \iota)$.

Before proving Theorem 5, I establish the following two helper lemmas.

Lemma 8 (framework-helper-1) *Let g and r be graphs such that r is a subgraph of g, let $cs \in D_c^*$, and let $lfp_g = S_{\mathcal{C}}(g, cs)$. Then:*

$$lfp_g \setminus Edges_r = S_{\mathcal{C}}(r, lfp_g(InEdges_r))$$

Proof

Let $lfp_r = S_{\mathcal{C}}(r, lfp_g(InEdges_r))$, so that we need to show $lfp_g \setminus Edges_r = lfp_r$.

Let FG_g be the global flow function FG from the definition of $S_{\mathcal{C}}(g, cs)$ (Definition 40).

Let FG_r be the global flow function FG from the definition of $S_{\mathcal{C}}(r, lfp_g(InEdges_r))$ (Definition 40).

Since r is a subgraph of g, any solution to $Eqs(g, cs)$, if restricted to the edges of r, will also be a solution to $Eqs(r, lfp_g(InEdges_r))$. As a result, if m is a fixed point of FG_g, then $m \setminus Edges_r$ is a fixed point of FG_r. Since lfp_g is the least fixed point of FG_g, it is a fixed point of FG_g, and therefore $lfp_g \setminus Edges_r$ is a fixed point of FG_r. Since lfp_r is the least fixed point of FG_r, we have $lfp_r \sqsubseteq lfp_g \setminus Edges_r$.

Because F_c is monotonic, we have that $\forall j \;.\; FGA_g^j(\bot_c) = FG_g^j(\bot_c)$ and $\forall j \;.\; FGA_r^j(\bot_c) = FG_r^j(\bot_c)$.

Therefore:

$$\begin{aligned} lfp_g &= \bigsqcup_{j=0}^{\infty} FG_g^j(\widetilde{\bot_c}) \\ lfp_r &= \bigsqcup_{j=0}^{\infty} FG_r^j(\widetilde{\bot_c}) \end{aligned} \tag{B.6}$$

If we can show that $\forall j \;.\; FG_g^j(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^j(\widetilde{\bot_c})$, then we are done, for then equations (B.6) would imply $lfp_g \setminus Edges_r \sqsubseteq lfp_r$, which combined with $lfp_r \sqsubseteq lfp_g \setminus Edges_r$, implies $lfp_g \setminus Edges_r = lfp_r$ (and this is what we had to show).

All we need to show now is that $\forall j \;.\; FG_g^j(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^j(\widetilde{\bot_c})$. We do this by induction on j.

- *Base case.* We have $FG_g^0(\widetilde{\bot_c}) = \widetilde{\bot_c}$, $FG_r^0(\widetilde{\bot_c}) = \widetilde{\bot_c}$, and so $FG_g^0(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^0(\widetilde{\bot_c})$

- *Inductive case.* We assume $FG_g^j(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^j(\widetilde{\bot_c})$, and we need to show:

  $$FG_g^{(j+1)}(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^{j+1}(\widetilde{\bot_c})$$

  Let $m_g = FG_g^j(\widetilde{\bot_c})$ and $m_r = FG_r^j(\widetilde{\bot_c})$. We therefore need to show:

  $$FG_g(m_g) \setminus Edges_r \sqsubseteq FG_r(m_r)$$

  which is:

  $$\forall e \in Edges_r \;.\; FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e)$$

  We pick $e \in Edges_r$, and show $FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e)$.

  There are two cases, based on whether or not e is an input edge of graph r:

  – Case where e is an input edge of graph r. Then there exists a k such that $InEdges_r[k]$. From the definition of FG_r (Definition 40), we have:

  $$\begin{aligned} FG_r(m_r)(e) &= \iota[k] \quad \text{(where } \iota \text{ is the tuple used to initialize the fixed point computation of } lfp_r\text{)} \\ &= lfp_g(InEdges_r)[k] \quad (\text{since } \iota = lfp_g(InEdges_r)) \\ &= lfp_g(e) \quad (\text{since } e = InEdges_r[k]) \end{aligned} \tag{B.7}$$

Because F_c is monotonic, $FG_g^0(\widetilde{\bot_c}), FG_g^1(\widetilde{\bot_c}), FG_g^2(\widetilde{\bot_c}), \ldots$ is an ascending chain, and so we have $\forall j \;.\; FG_g^j(\widetilde{\bot_c}) \sqsubseteq \mathit{lfp}_g$. As a result:

$$\begin{aligned}
& FG_g^{j+1}(\widetilde{\bot_c}) \sqsubseteq \mathit{lfp}_g \\
\Rightarrow\;& FG_g(m_g) \sqsubseteq \mathit{lfp}_g \qquad \text{(from the definition of } m_g\text{)} \\
\Rightarrow\;& FG_g(m_g)(e) \sqsubseteq \mathit{lfp}_g(e) \\
\Rightarrow\;& FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e) \qquad \text{(using (B.7))}
\end{aligned}$$

– Case where e is not an input edge of graph r. In this case, there exists a node n and an integer k such that $e = \mathit{out}_r(n)[k]$. From the definition of FG (Definition 40), we therefore have:

$$FG_r(m_r)(e) = F_c(n, \overrightarrow{m_r}(\mathit{in}_r(n)))[k] \tag{B.8}$$

Because r is a subgraph of g, we have $\mathit{in}_g(n) = \mathit{in}_r(n)$ and $\mathit{out}_g(n) = \mathit{out}_r(n)$. Then, by the definition of FG (Definition 40), we have:

$$FG_g(m_g)(e) = F_c(n, \overrightarrow{m_g}(\mathit{in}_r(n)))[k] \tag{B.9}$$

Recall that the inductive hypothesis is $m_g \setminus \mathit{Edges}_r \sqsubseteq m_r$. Since n is a node of r, the edges in the tuple $\mathit{in}_r(n)$ are all elements of Edges_r, and we can use this inductive hypothesis to get:

$$\begin{aligned}
& \overrightarrow{m_g}(\mathit{in}_r(n)) \sqsubseteq \overrightarrow{m_r}(\mathit{in}_r(n)) \\
\Rightarrow\;& F_c(n, \overrightarrow{m_g}(\mathit{in}_r(n)))[k] \sqsubseteq F_c(n, \overrightarrow{m_r}(\mathit{in}_r(n)))[k] \quad \text{(monotonicity of } F_c\text{)} \\
\Rightarrow\;& FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e) \quad \text{(using (B.8) and (B.9))}
\end{aligned}$$

■

Lemma 9 (framework-helper-2) *Let g be a graph, and let $n \in \mathit{Node}_g$ be a node in g. Let r be a replacement graph for n such that n and r have the same concrete semantics, or formally $\forall\; cs \in D_c^* \;.\; F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(\mathit{OutEdges}_r)$. Let g' be the graph resulting from replacing n by r in g, or $g' = \mathit{Update}(g, n, r)$. Then g and g' have the same concrete semantics, or:*

$$\forall cs \in D_c^* \;.\; \overrightarrow{S_{\mathcal{C}}(g, cs)}(\mathit{OutEdges}_g) = \overrightarrow{S_{\mathcal{C}}(g', cs)}(\mathit{OutEdges}_{g'})$$

Proof

Pick $cs \in D_c^*$, and show $\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) = \overrightarrow{S_{\mathcal{C}}(g', cs)}(OutEdges_{g'})$.

Let r' be the graph that is exactly the same as r, except that it's input and output edges are $in_g(n)$ and $out_g(n)$. The graph r' is the subgraph of g' that resulted from the substitution of n.

Let $\overrightarrow{x} = in_g(n)$, and $\overrightarrow{y} = out_g(n)$.

Let $lfp_g = S_{\mathcal{C}}(g, cs)$.

Let $cs_n = \overrightarrow{lfp_g}(\overrightarrow{x})$.

Let $lfp_{r'} = S_{\mathcal{C}}(r', cs_n)$.

Let $E = Edges_{g'} - Edges_g$. Intuitively, E is the set of edges in the replacement graph r, but without its input and output edges.

Let $m : Edge_{g'} \rightarrow D_c$ be defined as follows:

$$m(e) = \begin{cases} lfp_g(e) & \text{if } e \in Edges_g \\ lfp_{r'}(e) & \text{if } e \in E \end{cases} \tag{B.10}$$

m is a fixed point of $FG_{(g', cs)}$, since by construction it satisfies all the dataflow equations in $Eqs(g', cs)$.

The claim is that m is the least fixed point of $FG_{(g', cs)}$, which means that $m = S_{\mathcal{C}}(g', cs)$.

If this is the case, then because $OutEdges_g = OutEdges_{g'}$ and because all edges in $OutEdges_g$ are in $Edges_g$, we would have from Equation (B.10):

$$\overrightarrow{m}(OutEdges_{g'}) = \overrightarrow{lfp_g}(OutEdges_g)$$

which, since $m = S_{\mathcal{C}}(g', cs)$ and $lfp_g = S_{\mathcal{C}}(g, cs)$, becomes:

$$\overrightarrow{S_{\mathcal{C}}(g', cs)}(OutEdges_{g'}) = \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

which is what we had to show.

All we need to show now is that m is indeed the least fixed point of $FG_{(g', cs)}$.

Let m' be the least fixed point of $FG_{(g',cs)}$, so that $m' = S_{\mathcal{C}}(g', cs)$.

Since m' is the least fixed point of $FG_{(g',cs)}$, and since m is a fixed point of $FG_{(g',cs)}$, we must have $m' \sqsubseteq m$. We will now show $m \sqsubseteq m'$, which will establish that $m = m'$ and that m is indeed the least fixed point.

Since the domain of m and m' is $Edges_g \cup E$, to prove $m \sqsubseteq m'$, we will show $m \setminus Edges_g \sqsubseteq m' \setminus Edges_g$, and then $m \setminus E \sqsubseteq m' \setminus E$.

- Proof of $m \setminus Edges_g \sqsubseteq m' \setminus Edges_g$

  By Lemma 8, instantiated with $g = g'$, $r = r'$, we get that:

  $$m' \setminus Edges_{r'} = S_{\mathcal{C}}(r', \overrightarrow{m'}(InEdges_{r'}))$$

  By the construction of how n gets replaced with r in g, we have $\overrightarrow{x} = InEdges_{r'}$, and so we get:

  $$m' \setminus Edges_{r'} = S_{\mathcal{C}}(r', \overrightarrow{m'}(\overrightarrow{x})) \tag{B.11}$$

  Since $OutEdges_{r'} \subseteq Edge_{r'}$, we get:

  $$\overrightarrow{m'}(OutEdges_{r'}) = \overrightarrow{S_{\mathcal{C}}(r', \overrightarrow{m'}(\overrightarrow{x}))}(OutEdges_{r'})$$

  By the construction of how n gets replaced with r in g, we have $\overrightarrow{y} = OutEdges_{r'}$, and so we get:

  $$\overrightarrow{m'}(\overrightarrow{y}) = \overrightarrow{S_{\mathcal{C}}(r', \overrightarrow{m'}(\overrightarrow{x}))}(\overrightarrow{y}) \tag{B.12}$$

  From our assumptions, we know:

  $$\forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$$

  which, because r and r' only differ in their incoming and outgoing edges, gives us:

  $$\begin{aligned}
  & \forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r', cs)}(OutEdges_{r'}) \\
  \Rightarrow\ & \forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r', cs)}(\overrightarrow{y}) \qquad (\text{since } \overrightarrow{y} = OutEdges_{r'}) \\
  \Rightarrow\ & F_c(n, \overrightarrow{m'}(\overrightarrow{x})) = \overrightarrow{S_{\mathcal{C}}(r', \overrightarrow{m'}(\overrightarrow{x}))}(\overrightarrow{y}) \qquad (\text{instantiating with } cs = \overrightarrow{m'}(\overrightarrow{x})) \\
  \Rightarrow\ & F_c(n, \overrightarrow{m'}(\overrightarrow{x})) = \overrightarrow{m'}(\overrightarrow{y}) \qquad (\text{using (B.12)})
  \end{aligned} \tag{B.13}$$

Since $\overrightarrow{x} = in_g(n)$ and $\overrightarrow{y} = out_g(n)$, Equation (B.13) states that m' satisfies the local dataflow constraint for n in the equation $Eqs(g, cs)$.

Now consider a node $n' \in N_g$ different from n. Since m' is a solution to $Eqs(g', cs)$, and since the only difference between g and g' is that n was replaced with r, it must be the case that m' satisfies the local dataflow constraint for n' in the equation $Eqs(g, cs)$. Therefore, m' satisfies the local dataflow constraints of all nodes $n' \in N_g$ in the equation $Eqs(g, cs)$. As a result, m' is a solution to $Eqs(g, cs)$, and therefore:

$$
\begin{array}{lcll}
 & m' \setminus Edges_g \text{ is a fixed point of } FG_{(g,cs)} & & \\
\Rightarrow & lfp_g \sqsubseteq m' \setminus Edges_g & \text{(by the defn of least fixed point)} & \\
\Rightarrow & m \setminus Edges_g \sqsubseteq m' \setminus Edges_g & \text{(since for } \forall e \in Edges_g \,.\, m(e) = lfp_g(e)\text{)} & \text{(B.14)}
\end{array}
$$

- Proof of $m \setminus E \sqsubseteq m' \setminus E$

  Let $cs'_n = \overrightarrow{m'}(\overrightarrow{x})$. Since all the edges in the $\overrightarrow{x}$ tuple are in $Edges_g$, we get from (B.14):

$$
\begin{array}{lcll}
 & \overrightarrow{m}(\overrightarrow{x}) \sqsubseteq \overrightarrow{m'}(\overrightarrow{x}) & & \\
\Rightarrow & \overrightarrow{lfp_g}(\overrightarrow{x}) \sqsubseteq \overrightarrow{m'}(\overrightarrow{x}) & \text{(by (B.10) and the fact that } \overrightarrow{x} \in Edges_g^*\text{)} & \\
\Rightarrow & cs_n \sqsubseteq cs'_n & \text{(using definitions of } cs_n \text{ and } cs'_n\text{)} & \\
\Rightarrow & S_{\mathcal{C}}(r', cs_n) \sqsubseteq S_{\mathcal{C}}(r', cs'_n) & \text{(by the monotonicity of } S_{\mathcal{C}}\text{)} & \\
\Rightarrow & lfp_{r'} \sqsubseteq S_{\mathcal{C}}(r', cs'_n) & \text{(by defn of } lfp_{r'}\text{)} & \\
\Rightarrow & lfp_{r'} \sqsubseteq m' \setminus Edges_{r'} & \text{(by (B.11))} & \\
\Rightarrow & lfp_{r'} \setminus E \sqsubseteq m' \setminus E & \text{(since } E \subseteq Edges_{r'}\text{)} & \\
\Rightarrow & m \setminus E \sqsubseteq m' \setminus E & \text{(by (B.10))} &
\end{array}
$$

■

In order to prove Theorem 5, I will need a *measure* : $D_c^* \to \mathbb{N}$ function that decreases when a procedure call is made. I use the measure function to perform a proof by induction over all program states, and the fact that *measure* decreases at procedure calls allows me to assume inductively that the callee satisfies the property that I am trying to prove of the caller.

Unfortunately, a different measure function must be defined for each concrete domain that the analysis framework from Chapter 4 is instantiated with. One can think of the *measure* function as being part of the definition of the concrete semantics.

The *measure* function must satisfy the following requirements:

$$\forall \iota \in D_c^* \;.\; \iota = \overrightarrow{\bot_c} \Leftrightarrow measure(\iota) = 0 \tag{B.15}$$

$$\forall \iota_1 \in D_c^*, \iota_2 \in D_c^* \;.\; \iota_1 \sqsubseteq_c \iota_2 \Rightarrow measure(\iota_1) \leq measure(\iota_2) \tag{B.16}$$

$$\begin{aligned} &\forall \iota \in D_c^*, g \in Graph, n \in N_g \;. \\ &\quad [stmtAt(n) = (x := f(y)) \wedge \iota \neq \overrightarrow{\bot_c}] \Rightarrow \\ &\qquad measure(\iota) > measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g, \iota)}(in_g(n)))) \end{aligned} \tag{B.17}$$

Condition (B.15) simply states that *measure* returns 0 when evaluated on bottom, and that the only value for which *measure* returns 0 is bottom. Condition (B.16) states that the *measure* function is monotonic. Finally, condition (B.17) states that the *measure* function decreases at procedure calls.

The definition of *measure* uses an auxiliary function $remainingStackFrames : State \rightarrow \mathbb{N}$. Given a program state η, $remainingStackFrames(\eta)$ returns the number of stack frames remaining in η. It is defined as follows:

$$remainingStackFrames((\rho, \sigma, (f_1, \ldots, f_i), \mathcal{M})) = maxStackDepth - i$$

Note that since $0 \leq i \leq maxStackDepth$, it must be the case that:

$$\forall \eta \;.\; 0 \leq remainingStackFrames(\eta) \leq maxStackDepth$$

The concrete domain D_c is 2^{State} for both the forward and the backward concrete semantics. The *measure* function is defined identically in both cases:

$$measure((\eta s_1, \ldots, \eta s_i)) = \begin{cases} 0 & \text{if } \forall k \in [1..i] \;.\; \eta s_k = \bot_c \\ \max\limits_{k \in [1..i]} \; \max\limits_{\eta \in \eta s_k} (1 + remainingStackFrames(\eta)) & \text{otherwise} \end{cases}$$

The above *measure* function returns the maximum number of remaining stack frames in any of the program states in any of the ηs sets. The $\max_{\eta \in \eta s_k}$ operator is well defined

because *remainingStackFrames* is bounded by *maxStackDepth*. If *remainingStackFrames* were *not* bounded by *maxStackDepth*, then the $\max_{\eta \in \eta s_k}$ operator could return ∞, and this would cause the *measure* function not to satisfy (B.17), since both $measure(\iota)$ and $measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))))$ could be ∞.

Properties (B.15) and (B.16) hold trivially of the above *measure* function. The proof of property (B.17) is more complex, and is shown below.

Lemma 10 *The above measure function satisfies* (B.17)*.*

Proof

Pick $\iota \in D_c^*$, $g \in Graph$, and $n \in N_g$, assume $stmtAt(n) = (x := f(y)) \wedge \iota \neq \overrightarrow{\bot_c}$, and show $measure(\iota) > measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))))$.

There are two cases:

- $callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))) = \overrightarrow{\bot}$. In this case, from (B.15), $measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n)))) = 0$. Furthermore, from $\iota \neq \overrightarrow{\bot_c}$ and (B.15), we know that $measure(\iota) > 0$, which means that $measure(\iota) > measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))))$

- $callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))) \neq \overrightarrow{\bot}$. In the forward case, *callerToCallee* steps into the function call at node n, which will decrease *remainingStackFrames* on every program state, and thus cause *measure* to decrease. As a result, $measure(\overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))) > measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))))$. Furthermore, since the application of the forward flow function F_c does not modify the stack of any program states (even for function calls, since the function call is stepped over), we get: $measure(\iota) = measure(\overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n)))$, which then gives $measure(\iota) > measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))))$.

  In the backward case, *callerToCallee* is the identity function, and so $measure(\overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))) = measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))))$. The first statement in the CFG is a `return` statement (recall that backward analyses

run over the reverse CFG). Because executing F_c on a `return` statement causes *remainingStackFrames* to decrease, and running F_c on all other statement types does not modify the stack, we get that $measure(\iota) > measure(\overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n)))$, which then gives $measure(\iota) > measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g,\iota)}(in_g(n))))$.

■

Finally, I am now ready to prove Theorem 5.

Theorem 5 *Given an AT-analysis $(\mathcal{A}, R)$, where $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$, if F_a and R are sound, then $(\mathcal{A}, R)$ is sound.*

Proof

Without loss of generality, we can assume that the replacement function R always returns graph replacements with only one node in them. Indeed, if R returned a multiple node replacement graph r, then one can build a new replacement function R_{single} which, instead of returning r, returns a graph replacement containing a single node n equivalent to r (in that it satisfies $\forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$). Because R is sound, so would R_{single}, and the proof would proceed with R_{single} instead of R. Lemma 9 can then be used to argue that the soundness of $(\mathcal{A}, R_{single})$ implies the soundness of $(\mathcal{A}, R)$.

Let $(\mathcal{A}, R)$ be an AT-analysis that is locally sound, let $\pi = (p_1, \ldots, p_n)$ be a program where $g_i = cfg(p_i)$, and let $[\![\mathcal{A}, R]\!](\pi) = \pi'$ where $\pi' = (p'_1, \ldots, p'_n)$ and $r_i = cfg(p'_i)$. We want to show:

$$\forall i \in [1..n]\ .\ \forall \iota_c \in D_c^*\ .\ \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(OutEdges_{r_i})$$

We let Int_{g_i}, FG_{g_i}, and FGA_{g_i} be the interpretation function, the global flow function, and the global ascending flow function in the definition of $S_{\mathcal{C}}(g_i, \iota_c)$.

We let Int_{r_i}, FG_{r_i}, and FGA_{r_i} be the interpretation function, the global flow function, and the global ascending flow function in the definition of $S_{\mathcal{C}}(r_i, \iota_c)$.

For $l \geq 0$, let $P(l)$ be the following formula:

$$P(l) = \quad \forall i \in [1..n] \;.\; \forall \iota_c \in D_c^* \;. \\ \mathit{measure}(\iota_c) \leq l \Rightarrow \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\mathit{OutEdges}_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(\mathit{OutEdges}_{r_i})$$

Our goal is to prove $\forall l \geq 0 \;.\; P(l)$. We do this by induction on l.

- *Base case.* We need to show $P(0)$.

  Pick $i \in [1..n]$, $\iota_c \in D_c^*$, assume:

  $$\mathit{measure}(\iota_c) \leq 0 \tag{B.18}$$

  and show:

  $$\overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\mathit{OutEdges}_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(\mathit{OutEdges}_{r_i})$$

  Because *measure* always returns a value greater or equal to 0, from (B.18), we get that $\mathit{measure}(\iota_c) = 0$. Furthermore, since the only ι_c for which *measure* can return 0 is $\overrightarrow{\bot_c}$, we get that $\iota_c = \overrightarrow{\bot_c}$. Since F_c is bottom-preserving, we get that $\overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\mathit{OutEdges}_{g_i}) = \overrightarrow{\bot_c}$, and so then we trivially get:

  $$\overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\mathit{OutEdges}_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(\mathit{OutEdges}_{r_i})$$

- *Inductive case.* We assume $P(l)$ and show $P(l+1)$, which is:

  $$\forall i \in [1..n] \;.\; \forall \iota_c \in D_c^* \;. \\ \mathit{measure}(\iota_c) \leq l+1 \Rightarrow \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\mathit{OutEdges}_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(\mathit{OutEdges}_{r_i})$$

  Pick $i \in [1..n]$, $\iota_c \in D_c^*$, assume:

  $$\mathit{measure}(\iota_c) \leq l+1 \tag{B.19}$$

  and show:

  $$\overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\mathit{OutEdges}_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(\mathit{OutEdges}_{r_i})$$

  If $\iota_c = \overrightarrow{\bot_c}$, then we immediately get this, using the argument from the base case. Therefore we only need to consider the case where $\iota_c \neq \overrightarrow{\bot_c}$.

Using the definition of $S_{\mathcal{C}}$ we need to show:

$$\bigsqcup_{j=0}^{\infty} FGA^j_{g_i}(\widetilde{\bot_c}) \sqsubseteq_c \bigsqcup_{j=0}^{\infty} FGA^j_{r_i}(\widetilde{\bot_c})$$

Since F_c is monotonic, we have that $FG^j_{r_i}(\widetilde{\bot_c})$ and $FG^j_{g_i}(\widetilde{\bot_c})$ are ascending chains, and so:

$$\forall j \geq 0. FG^j_{g_i}(\widetilde{\bot_c}) = FGA^j_{g_i}(\widetilde{\bot_c})$$
$$\forall j \geq 0. FG^j_{r_i}(\widetilde{\bot_c}) = FGA^j_{r_i}(\widetilde{\bot_c})$$

Thus, all we need to show is:

$$\bigsqcup_{j=0}^{\infty} FG^j_{g_i}(\widetilde{\bot_c}) \sqsubseteq_c \bigsqcup_{j=0}^{\infty} FG^j_{r_i}(\widetilde{\bot_c})$$

To do this, we show:

$$\forall j \geq 0 \ . \ FG^j_{g_i}(\widetilde{\bot_c}) \sqsubseteq_c FG^j_{r_i}(\widetilde{\bot_c})$$

We do this by induction on j.

– *Base case.* We need to show $FG^0_{g_i}(\widetilde{\bot_c}) \sqsubseteq_c FG^0_{r_i}(\widetilde{\bot_c})$. This follows immediately from the fact that $FG^0_{g_i}(\widetilde{\bot_c}) = \widetilde{\bot_c}$ and $FG^0_{r_i}(\widetilde{\bot_c}) = \widetilde{\bot_c}$.

– *Inductive case.* We assume $FG^j_{g_i}(\widetilde{\bot_c}) \sqsubseteq_c FG^j_{r_i}(\widetilde{\bot_c})$, and we need to show $FG^{j+1}_{g_i}(\widetilde{\bot_c}) \sqsubseteq_c FG^{j+1}_{r_i}(\widetilde{\bot_c})$.

Let $a = FG^j_{g_i}(\widetilde{\bot_c})$ and $b = FG^j_{r_i}(\widetilde{\bot_c})$, so that we are assuming:

$$a \sqsubseteq_c b \tag{B.20}$$

We need to show $FG_{g_i}(a) \sqsubseteq_c FG_{r_i}(b)$, which, because $Edges_{g_i} = Edges_{r_i}$, is:

$$\forall e \in Edges_{g_i} \ . \ FG_{g_i}(a)(e) \sqsubseteq_c FG_{r_i}(b)(e)$$

which is:

$$\forall e \in Edges_{g_i} \ . \ Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b)$$

Let $e \in Edges_{g_i}$, and we need to show:

$$Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b) \tag{B.21}$$

There are four cases, based on the definition of $Int(e, m)$, and on how n was transformed:

* There is some integer k such that $e = InEdges[k]$

  In this case, $Int_{g_i}(e, a) = \iota_c[k]$ and $Int_{r_i}(e, b) = \iota_c[k]$, and so $Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b)$.

* There is some integer k and some node n such that $e = out_{g_i}(n)[k]$, and n was not modified from g_i to r_i and n is not a call.

  Let n' be the corresponding node to n in r_i. Recall that if a node is not modified from g_i to r_i, it is actually copied over, with to a new node which has the same statement as the original node. As a result, we have $stmtAt(n) = stmtAt(n')$.

  By the definition of Int, we get $Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(in_{g_i}(n)))[k]$.

  By the definition of T, we get that $e = out_{r_i}(n')[k]$, and so by the definition of Int, we get $Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(in_{r_i}(n')))[k]$.

  By the definition of T, we get that $in_{g_i}(n) = in_{r_i}(n')$. Let $\overrightarrow{x} = in_{g_i}(n)$. Thus:

$$Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \tag{B.22}$$

$$Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k] \tag{B.23}$$

  Because n is not a call, F_c evaluated at n only depends on $stmtAt(n)$. Since $stmtAt(n) = stmtAt(n')$, we therefore have:

$$F_c(n, \overrightarrow{b}(\overrightarrow{x}))[k] = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k] \tag{B.24}$$

  From (B.20), we know that:

$$\begin{aligned}
& a \sqsubseteq_c b \\
\Rightarrow\quad & \overrightarrow{a}(\overrightarrow{x}) \sqsubseteq_c \overrightarrow{b}(\overrightarrow{x}) \\
\Rightarrow\quad & F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \sqsubseteq_c F_c(n, \overrightarrow{b}(\overrightarrow{x}))[k] && \text{(by the monotonicity of } F_c\text{))} \\
\Rightarrow\quad & F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \sqsubseteq_c F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k] && \text{(using (B.24))} \\
\Rightarrow\quad & Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b) && \text{(using (B.22) and (B.23))}
\end{aligned}$$

  And this is what we had to show in (B.21).

* There is some integer k and some node n such that $e = out_{g_i}(n)[k]$, and n was not modified from g_i to r_i, and n is a call.

  Let n' be the corresponding node to n in r_i (recall that if a node is not modified from g_i to r_i, it is actually copied over, with to a new node which has the same statement as

the original node). Because n is a call, we therefore have $stmtAt(n) = stmtAt(n') = (y := f(z))$

By the definition of Int, we get $Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(in_{g_i}(n)))[k]$.

By the definition of T, we get that $e = out_{r_i}(n')[k]$, and so by the definition of Int, we get $Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(in_{r_i}(n')))[k]$.

By the definition of T, we get that $in_{g_i}(n) = in_{r_i}(n')$. Let $\overrightarrow{x} = in_{g_i}(n)$. Thus:

$$Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k]$$
$$Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k]$$

Since both n and n' are call nodes, they only have one successor in the intraprocedural CFG (which is the CFG over which we are reasoning), and so it must be the case that $k = 0$. Thus:

$$Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[0]$$
$$Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[0]$$

Let $g_u = cfg(callee(n))$ and $r_u = cfg(callee(n'))$. The index u matches because n and n' are calling the same function.

From the definition of F_c in Equation (5), we get:

$$Int_{g_i}(e, a) = calleeToCaller(n, \overrightarrow{S_{\mathcal{C}}(g_u, \iota_a)}(OutEdges_{g_u})) \quad \text{where } \iota_a = callerToCallee(n, \overrightarrow{a}(\overrightarrow{x})) \tag{B.25}$$

$$Int_{r_i}(e, b) = calleeToCaller(n', \overrightarrow{S_{\mathcal{C}}(r_u, \iota_b)}(OutEdges_{r_u})) \quad \text{where } \iota_b = callerToCallee(n', \overrightarrow{b}(\overrightarrow{x})) \tag{B.26}$$

Since we are only considering the case where $\iota \neq \overrightarrow{\bot_c}$, we can use (B.17) to get:

$$measure(\iota_c) > measure(callerToCallee(n, \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\overrightarrow{x}))) \tag{B.27}$$

Since $a = FG^j_{g_i}(\widetilde{\bot_c})$, and since $S_{\mathcal{C}}(g_i, \iota_c) = \bigsqcup_{j=0}^{\infty} FG^j_{g_i}(\widetilde{\bot_c})$, we have by the definition

of $\sqcup$ that:

$$
\begin{aligned}
& a \sqsubseteq_c S_{\mathcal{C}}(g_i, \iota_c) \\
\Rightarrow \quad & \overrightarrow{a}(\overrightarrow{x}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\overrightarrow{x}) \\
\Rightarrow \quad & \mathit{callerToCallee}(n, \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c \mathit{callerToCallee}(n, \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\overrightarrow{x})) \\
& \text{(by monotonicity of } \mathit{callerToCallee} \text{ from Equation (4.3))} \\
\Rightarrow \quad & \iota_a \sqsubseteq_c \mathit{callerToCallee}(n, \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\overrightarrow{x})) \\
& \text{(by definition of } \iota_a) \\
\Rightarrow \quad & \mathit{measure}(\iota_a) \leq \mathit{measure}(\mathit{callerToCallee}(n, \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\overrightarrow{x}))) \\
& \text{(by monotonicity of } \mathit{measure}) \\
\Rightarrow \quad & \mathit{measure}(\iota_c) > \mathit{measure}(\iota_a) \\
& \text{(from (B.27) and the fact that } a > b \geq c \text{ implies } a > c) \\
\Rightarrow \quad & \mathit{measure}(\iota_a) \leq l \\
& \text{(from (B.19) and the fact that } a < b \leq l + 1 \text{ implies } a \leq l)
\end{aligned}
$$

From our inductive hypothesis, we know $P(l)$, which is:

$$
\begin{aligned}
P(l) = \forall i \in [1..n] \; . \; & \forall \iota_c \in D_c^* \; . \\
& \mathit{measure}(\iota_c) \leq l \Rightarrow \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(\mathit{OutEdges}_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(\mathit{OutEdges}_{r_i})
\end{aligned}
$$

We instantiate this with $i = u$ and $\iota_c = \iota_a$ to get:

$$
\overrightarrow{S_{\mathcal{C}}(g_u, \iota_a)}(\mathit{OutEdges}_{g_u}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_u, \iota_a)}(\mathit{OutEdges}_{r_u}) \tag{B.28}
$$

From (B.20), we have:

$a \sqsubseteq_c b$

$\Rightarrow$ $\overrightarrow{a}(\overrightarrow{x}) \sqsubseteq_c \overrightarrow{b}(\overrightarrow{x})$

$\Rightarrow$ $callerToCallee(n, \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c callerToCallee(n, \overrightarrow{b}(\overrightarrow{x}))$

(by the monotonicity of *callerToCallee* from Equation (4.3))

$\Rightarrow$ $callerToCallee(n, \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c callerToCallee(n', \overrightarrow{b}(\overrightarrow{x}))$

(from Equation (4.1), combined with $stmtAt(n) = stmtAt(n')$)

$\Rightarrow$ $\iota_a \sqsubseteq_c \iota_b$

(from the definition of ι_a and ι_b)

$\Rightarrow$ $\overrightarrow{S_{\mathcal{C}}(r_u, \iota_a)}(OutEdges_{r_u}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_u, \iota_b)}(OutEdges_{r_u})$

(from the monotonicity of $S_{\mathcal{C}}$, which follows from the monotonicity of F_c)

$\Rightarrow$ $\overrightarrow{S_{\mathcal{C}}(g_u, \iota_a)}(OutEdges_{g_u}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_u, \iota_b)}(OutEdges_{r_u})$

(from Equation (B.28) and the transitivity of $\sqsubseteq_c$)

$\Rightarrow$ $calleeToCaller(n, \overrightarrow{S_{\mathcal{C}}(g_u, \iota_a)}(OutEdges_{g_u})) \sqsubseteq_c$
$calleeToCaller(n, \overrightarrow{S_{\mathcal{C}}(r_u, \iota_b)}(OutEdges_{r_u}))$

(by the monotonicity of *calleeToCaller* from Equation (4.4))

$\Rightarrow$ $calleeToCaller(n, \overrightarrow{S_{\mathcal{C}}(g_u, \iota_a)}(OutEdges_{g_u})) \sqsubseteq_c$
$calleeToCaller(n', \overrightarrow{S_{\mathcal{C}}(r_u, \iota_b)}(OutEdges_{r_u}))$

(from Equation (4.2), combined with $stmtAt(n) = stmtAt(n')$)

$\Rightarrow$ $Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b)$

(from equations (B.25) and (B.26))

And this is what we had to show in (B.21)

$*$ There is some integer k and some node n such that $e = out_{g_i}(n)[k]$, and n in g_i was modified to n' in r_i because R returned a single-node replacement graph r_n for n.
Then by the definition of Int, we get $Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(in_{g_i}(n)))[k]$.
Since n was modified to a single-node graph n', by the definition of T, we get that $e = out_{r_i}(n')[k]$, and so by the definition of Int, we get $Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(in_{r_i}(n')))[k]$.

Since n was modified to a single-node graph n', by the definition of T, we get that $in_{g_i}(n) = in_{r_i}(n')$. Let $\overrightarrow{x} = in_{g_i}(n)$. Thus:

$$Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \tag{B.29}$$

$$Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k] \tag{B.30}$$

Let FGA_a be the global ascending function from the definition of $S_{\mathcal{A}}(g_i, \top_a)$.
Since $\alpha(\iota_c) \sqsubseteq_a \top_a$, we have from the proof of Theorem 4:

$$\forall j \geq 0 \,.\, \widetilde{\alpha}(FG^j_{g_i}(\widetilde{\bot_c})) \sqsubseteq_a FGA^j_a(\widetilde{\bot_a})$$

Instantiating this with the current j, we get::

$$\widetilde{\alpha}(FG^j_{g_i}(\widetilde{\bot_c})) \sqsubseteq_a FGA^j_a(\widetilde{\bot_a})$$

or, equivalently:

$$\widetilde{\alpha}(a) \sqsubseteq_a FGA^j_a(\widetilde{\bot_a})$$

Furthermore, from the definition of $S_{\mathcal{A}}$ and $\sqcup_a$, we have that:

$$FGA^j_a(\widetilde{\bot_a}) \sqsubseteq_a S_{\mathcal{A}}(g_i, \top_a)$$

Thus, by transitivity:

$$\widetilde{\alpha}(a) \sqsubseteq_a S_{\mathcal{A}}(g_i, \top_a)$$

Let $m = S_{\mathcal{A}}(g_i, \top_a)$, so that:

$$\widetilde{\alpha}(a) \sqsubseteq_a m$$

Let $cs = \overrightarrow{a}(\overrightarrow{x})$ and let $ds = \overrightarrow{m}(\overrightarrow{x})$. Therefore $\overrightarrow{\alpha}(cs) \sqsubseteq_a ds$.
From the definition of $[\![\mathcal{A}, R]\!]$ (Definition 6), we know that:

$$r_i = T(R, g_i, m)$$

For n to have been replaced with r_n, by the definition of T, it therefore must be the case that:

$$R(n, \overrightarrow{m}(\overrightarrow{x})) = r_n$$

From the soundness of R, we know that (4.6) holds. Instantiating (4.6) with $n = n, ds = ds, g = r_n, cs = cs$, and using $\overrightarrow{\alpha}(cs) \sqsubseteq_a ds$, we get:

$$F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_n, cs)}(OutEdges_{r_n})$$

Since r_n is a single-node graph with node n', we have that

$$\overrightarrow{S_{\mathcal{C}}(r_n, cs)}(OutEdges_{r_n}) = F_c(n', cs)$$

Thus:

$$\begin{array}{rl} & F_c(n, cs) \sqsubseteq_c F_c(n', cs) \\ \Rightarrow & F_c(n, \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c F_c(n', \overrightarrow{a}(\overrightarrow{x})) \end{array} \tag{B.31}$$

From(B.20), we have:

$$\begin{array}{lll} & a \sqsubseteq_c b & \\ \Rightarrow & \overrightarrow{a}(\overrightarrow{x}) \sqsubseteq_c \overrightarrow{b}(\overrightarrow{x}) & \\ \Rightarrow & F_c(n', \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c F_c(n', \overrightarrow{b}(\overrightarrow{x})) & \text{monotonicity of } F_c \\ \Rightarrow & F_c(n, \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c F_c(n', \overrightarrow{b}(\overrightarrow{x})) & \text{transitivity and (B.31)} \\ \Rightarrow & F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \sqsubseteq_c F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k] & \\ \Rightarrow & Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b) & \text{using (B.29) and (B.30)} \end{array}$$

And this is what we had to show in (B.21).

■

Appendix C

ADDITIONAL MATERIAL FOR FORWARD RHODIUM OPTIMIZATIONS

C.1 Proofs

Theorem 6 *If the syntactic form* ψ_1 `=>` ψ_2 *is disallowed, and the syntactic form* `!`ψ *is allowed only if* ψ *is an equality (i.e.* t_1 `==` t_2*), or an inequality (i.e.* t_1 `!=` t_2*) then* F_O *is monotonic.*

Proof

We assume the syntactic restrictions on the antecedents from Theorem 6 and we need to show:

$$\forall (n, ds_1, ds_2) \in Node \times D^* \times D^* \; . \; ds_1 \sqsubseteq ds_2 \Rightarrow F_O(n, ds_1) \sqsubseteq F_O(n, ds_2)$$

Or, equivalently:

$$\forall (n, ds_1, ds_2, h) \in Node \times D^* \times D^* \times \mathbb{N} \; . \; ds_1 \sqsubseteq ds_2 \Rightarrow F_O(n, ds_1)[h] \sqsubseteq F_O(n, ds_2)[h]$$

Using the definition of $\sqsubseteq$, we need to show:

$$\forall \; n, ds_1, ds_2, h \; . \; ds_1 \sqsubseteq ds_2 \Rightarrow F_O(n, ds_1)[h] \supseteq F_O(n, ds_2)[h]$$

Using the definition of F_O from Equation (5.3), it is sufficient to show:

$$\forall \; \psi, n, ds_1, ds_2, \theta \; . \; (ds_1 \sqsubseteq ds_2 \wedge [\![\psi]\!](\theta, ds_2, n)) \Rightarrow [\![\psi]\!](\theta, ds_1, n)$$

Let $P(\psi) = \forall \; n, ds_1, ds_2, \theta \; . \; (ds_1 \sqsubseteq ds_2 \wedge [\![\psi]\!](\theta, ds_2, n)) \Rightarrow [\![\psi]\!](\theta, ds_1, n)$. We need to show $\forall \psi \; . \; P(\psi)$. We do this by induction on the syntactic structure of ψ.

- Cases where ψ is `true`, `false`, t_1 `==` t_2, or t_1 `!=` t_2

  In all these cases $[\![\psi]\!](\theta, ds, n)$ does not depend on ds.

  Therefore $[\![\psi]\!](\theta, ds_1, n) = [\![\psi]\!](\theta, ds_2, n)$, and so $P(\psi)$ holds.

- Case `!`ψ

  We need to show $P(!\psi)$.

  Because of the syntactic restrictions mentioned in Theorem 6, this case only occurs with $\psi = (t_1 == t_2)$ or $\psi = (t_1 ! = t_2)$.

  In both of these cases, $[\![\psi]\!](\theta, ds, n)$ does not depend on ds, so that $[\![\psi]\!](\theta, ds_1, n) = [\![\psi]\!](\theta, ds_2, n)$.

  This implies $\neg[\![\psi]\!](\theta, ds_1, n) = \neg[\![\psi]\!](\theta, ds_2, n)$.

  By the definition of $[\![\cdot]\!](\theta, ds, n)$ this implies $[\![!\psi]\!](\theta, ds_1, n) = [\![!\psi]\!](\theta, ds_2, n)$.

  And therefore $P(!\psi)$ holds.

- Case ψ_1 `||` ψ_2

  By the induction hypothesis we know $P(\psi_1)$ and $P(\psi_2)$, and we need to show $P(\psi_1$ `||` $\psi_2)$.

  To show $P(\psi_1$ `||` $\psi_2)$, we assume:

  $$ds_1 \sqsubseteq ds_2 \wedge [\![\psi_1 \texttt{ || } \psi_2]\!](\theta, ds_2, n)$$

  and show:

  $$[\![\psi_1 \texttt{ || } \psi_2]\!](\theta, ds_1, n)$$

  By the definition of $[\![\cdot]\!](\theta, ds, n)$, we have:

  $$\begin{aligned} [\![\psi_1 \texttt{ || } \psi_2]\!](\theta, ds_2, n) &= [\![\psi_1]\!](\theta, ds_2, n) \vee [\![\psi_2]\!](\theta, ds_2, n) \\ [\![\psi_1 \texttt{ || } \psi_2]\!](\theta, ds_1, n) &= [\![\psi_1]\!](\theta, ds_1, n) \vee [\![\psi_2]\!](\theta, ds_1, n) \end{aligned}$$

Let $A = [\![\psi_1]\!](\theta, ds_2, n)$, $B = [\![\psi_2]\!](\theta, ds_2, n)$, $C = [\![\psi_1]\!](\theta, ds_1, n)$, and $D = [\![\psi_2]\!](\theta, ds_1, n)$.

With these definitions, we are assuming $ds_1 \sqsubseteq ds_2$ and $A \vee B$ and trying to show $C \vee D$. We do case analysis on whether or not A holds.

- Case where A holds. Then by $ds_1 \sqsubseteq ds_2$ and $P(\psi_1)$, we get that C holds, and so $C \vee D$.
- Case where A does not hold. Then B must hold. Then by $ds_1 \sqsubseteq ds_2$ and $P(\psi_2)$, we get that D holds, and so $C \vee D$.

- Case ψ_1 `&&` ψ_2

  By the induction hypothesis we know $P(\psi_1)$ and $P(\psi_2)$, and we need to show $P(\psi_1$ `&&` $\psi_2)$.

  To show $P(\psi_1$ `&&` $\psi_2)$, we assume:

  $$ds_1 \sqsubseteq ds_2 \wedge [\![\psi_1 \texttt{ \&\& } \psi_2]\!](\theta, ds_2, n)$$

  and show:

  $$[\![\psi_1 \texttt{ \&\& } \psi_2]\!](\theta, ds_1, n)$$

  By the definition of $[\![\cdot]\!](\theta, ds, n)$, we have:

  $$\begin{aligned} [\![\psi_1 \texttt{ \&\& } \psi_2]\!](\theta, ds_2, n) &= [\![\psi_1]\!](\theta, ds_2, n) \wedge [\![\psi_2]\!](\theta, ds_2, n) \\ [\![\psi_1 \texttt{ \&\& } \psi_2]\!](\theta, ds_1, n) &= [\![\psi_1]\!](\theta, ds_1, n) \wedge [\![\psi_2]\!](\theta, ds_1, n) \end{aligned}$$

  Let $A = [\![\psi_1]\!](\theta, ds_2, n)$, $B = [\![\psi_2]\!](\theta, ds_2, n)$, $C = [\![\psi_1]\!](\theta, ds_1, n)$, and $D = [\![\psi_2]\!](\theta, ds_1, n)$.

  With these definitions, we are assuming $ds_1 \sqsubseteq ds_2$ and $A \wedge B$ and trying to show $C \wedge D$.

  Because A holds, using $ds_1 \sqsubseteq ds_2$ and $P(\psi_1)$, we get that C holds.

  Because B holds, using $ds_1 \sqsubseteq ds_2$ and $P(\psi_2)$, we get that D holds.

  Thus $C \wedge D$ holds.

- Case $\psi_1 => \psi_2$

  Because of the syntactic restrictions mentioned in Theorem 6, this case does not occur.

- Case $\texttt{forall}\ X : \tau\ .\ \psi$

  By the induction hypothesis, we know $P(\psi)$, and we need to show $P(\texttt{forall}\ X : \tau\ .\ \psi)$.

  To show $P(\texttt{forall}\ X : \tau\ .\ \psi)$, we assume:

  $$ds_1 \sqsubseteq ds_2 \wedge [\![\texttt{forall}\ X : \tau\ .\ \psi]\!](\theta, ds_2, n)$$

  and show:

  $$[\![\texttt{forall}\ X : \tau\ .\ \psi]\!](\theta, ds_1, n)$$

  By the definition of $[\![\cdot]\!](\theta, ds, n)$, we have:

  $$\begin{aligned} [\![\texttt{forall}\ X : \tau\ .\ \psi]\!](\theta, ds_2, n) &= \forall\, t : \tau\ .\ [\![\psi]\!](\theta[X \mapsto t], ds_2, n) \\ [\![\texttt{forall}\ X : \tau\ .\ \psi]\!](\theta, ds_1, n) &= \forall\, t : \tau\ .\ [\![\psi]\!](\theta[X \mapsto t], ds_1, n) \end{aligned}$$

  Let $A = \forall\, t : \tau\ .\ [\![\psi]\!](\theta[X \mapsto t], ds_2, n)$ and $B = \forall\, t : \tau\ .\ [\![\psi]\!](\theta[X \mapsto t], ds_1, n)$.

  With these definitions, we are assuming $ds_1 \sqsubseteq ds_2$ and A and trying to show B.

  To show B, pick a $t : \tau$, and show $[\![\psi]\!](\theta[X \mapsto t], ds_1, n)$.

  Instantiating A with t, we get $[\![\psi]\!](\theta[X \mapsto t], ds_2, n)$.

  Using $ds_1 \sqsubseteq ds_2$ and $P(\psi)$, we get $[\![\psi]\!](\theta[X \mapsto t], ds_1, n)$ (Note that the θ in the $P(\psi)$ quantifier gets instantiated with $\theta[X \mapsto t]$).

- Case $\texttt{exists}\ X : \tau\ .\ \psi$

  By the induction hypothesis, we know $P(\psi)$, and we need to show $P(\texttt{exists}\ X : \tau\ .\ \psi)$.

  To show $P(\texttt{exists}\ X : \tau\ .\ \psi)$, we assume:

  $$ds_1 \sqsubseteq ds_2 \wedge [\![\texttt{exists}\ X : \tau\ .\ \psi]\!](\theta, ds_2, n)$$

  and show:

  $$[\![\texttt{exists}\ X : \tau\ .\ \psi]\!](\theta, ds_1, n)$$

By the definition of $[\![\cdot]\!](\theta, ds, n)$, we have:

$$\begin{aligned}
[\![\texttt{exists } X : \tau \ . \ \psi]\!](\theta, ds_2, n) &= \exists\, t : \tau \ . \ [\![\psi]\!](\theta[X \mapsto t], ds_2, n) \\
[\![\texttt{exists } X : \tau \ . \ \psi]\!](\theta, ds_1, n) &= \exists\, t : \tau \ . \ [\![\psi]\!](\theta[X \mapsto t], ds_1, n)
\end{aligned}$$

Let $A = \exists\, t : \tau \ . \ [\![\psi]\!](\theta[X \mapsto t], ds_2, n)$ and $B = \exists\, t : \tau \ . \ [\![\psi]\!](\theta[X \mapsto t], ds_1, n)$.

With these definitions, we are assuming $ds_1 \sqsubseteq ds_2$ and A and trying to show B.

From A we know there exists t such that $[\![\psi]\!](\theta[X \mapsto t], ds_2, n)$.

Using $ds_1 \sqsubseteq ds_2$ and $P(\psi)$, we get $[\![\psi]\!](\theta[X \mapsto t], ds_1, n)$ (Note that the θ in the $P(\psi)$ quantifier gets instantiated with $\theta[X \mapsto t]$).

Thus B holds, with t being the witness to the existential quantifier in B.

- Case $EF(t_1, \ldots, t_i)\texttt{in}[h]$

  We need to show $P(EF(t_1, \ldots, t_i)@\texttt{in}[h])$.

  To show this, assume:

  $$ds_1 \sqsubseteq ds_2 \wedge [\![EF(t_1, \ldots, t_i)@\texttt{in}[h]]\!](\theta, ds_2, n)$$

  and show:

  $$[\![EF(t_1, \ldots, t_i)@\texttt{in}[h]]\!](\theta, ds_1, n)$$

  By the definition of $[\![\cdot]\!](\theta, ds, n)$, we have:

  $$\begin{aligned}
  [\![EF(t_1, \ldots, t_i)@\texttt{in}[h]]\!](\theta, ds_2, n) &= EF([\![t_1]\!](\theta, n), \ldots, [\![t_i]\!](\theta, n)) \in ds_2[h] \\
  [\![EF(t_1, \ldots, t_i)@\texttt{in}[h]]\!](\theta, ds_1, n) &= EF([\![t_1]\!](\theta, n), \ldots, [\![t_i]\!](\theta, n)) \in ds_1[h]
  \end{aligned}$$

  Let $A = EF([\![t_1]\!](\theta, n), \ldots, [\![t_i]\!](\theta, n)) \in ds_2[h]$ and $B = EF([\![t_1]\!](\theta, n), \ldots, [\![t_i]\!](\theta, n)) \in ds_1[h]$.

  With these definitions, we are assuming $ds_1 \sqsubseteq ds_2$ and A and trying to show B.

  Since $ds_1 \sqsubseteq ds_2$, we have $ds_1[h] \sqsubseteq ds_2[h]$.

By the definition of $\sqsubseteq$, we have $ds_1[h] \supseteq ds_2[h]$.

$ds_1[h] \supseteq ds_2[h]$ combined with A then gives B.

■

Lemma 2 *The forward abstraction function α from Definition 18 is join-monotonic.*

Proof

We need to show: $\alpha(\bigsqcup_c Y) \sqsubseteq \bigsqcup\{\alpha(c) \mid c \in Y\}$, where Y is any chain of the lattice domain D_c.

For our lattice, we will prove the following stronger property:

$$\forall Y \in 2^{D_c} \;.\; \alpha(\bigsqcup\nolimits_c Y) \sqsubseteq \bigsqcup\{\alpha(c) \mid c \in Y\} \tag{C.1}$$

This property is stronger because it holds for any set Y of D_c elements, not only for chains.

To show (C.1), pick $Y \in 2^{D_c}$, and prove:

$$\alpha(\bigsqcup\nolimits_c Y) \sqsubseteq \bigsqcup\{\alpha(c) \mid c \in Y\}$$

Using the definition of $\sqcup_c$ and $\sqcup$, this becomes:

$$\alpha(\bigcup Y) \supseteq \bigcap\{\alpha(c) \mid c \in Y\}$$

Let $Q = \bigcup Y$ and let $R = \bigcap\{\alpha(c) \mid c \in Y\}$. Note that because the type of α is $\alpha : D_c \to D$, we have that R is a set of elements of D, and therefore $R \subseteq D$.

With these definitions, we now need to show $\alpha(Q) \supseteq R$. To do this, assume $f \in R$ and show $f \in \alpha(Q)$. From $R \subseteq D$, $f \in R$, and the definition of D (Definition 14), we get:

$$f = EF(t_1, \ldots, t_i) \tag{C.2}$$

$$EF \in EdgeFact_O \wedge i = arity(EF) \wedge (t_1, \ldots, t_i) \in GroundTerm^i \tag{C.3}$$

Our assumption is therefore $EF(t_1, \ldots, t_i) \in R$, and we are trying to show $EF(t_1, \ldots, t_i) \in$

$\alpha(Q)$. To do this, by the definition of α (Definition 18), we need to show:

$$EF \in EdgeFact_O \wedge i = arity(EF) \wedge (t_1, \ldots, t_i) \in GroundTerm^i \tag{C.4}$$

$$\forall \eta \in Q \; . \; [\![EF]\!](t_1, \ldots, t_i, \eta) \tag{C.5}$$

Condition (C.4) is already provided from (C.3). So we are only left with showing (C.5).

From the assumption introduced to show $\alpha(Q) \supseteq R$, we have $EF(t_1, \ldots, t_i) \in R$, and since $R = \bigcap\{\alpha(c) \mid c \in Y\}$, we get from the definition of $\bigcap$:

$$\forall c \in Y \; . \; EF(t_1, \ldots, t_i) \in \alpha(c)$$

which, from the definition of α (Definition 18), gives us:

$$\forall c \in Y \; . \; \forall \eta \in c \; . \; [\![EF]\!](t_1, \ldots, t_i, \eta) \tag{C.6}$$

Now we can show (C.5). To do this, assume $\eta \in Q$, and show $[\![EF]\!](t_1, \ldots, t_i, \eta)$.

Since $Q = \bigcup Y$, from $\eta \in Q$ and the definition of $\bigcup$, we get that there exists a c such that $c \in Y \wedge \eta \in c$. Instantiating (C.6) with c and η, we get $[\![EF]\!](t_1, \ldots, t_i, \eta)$, which is what we had to show.

■

Before proving Lemmas 3 and 4, I first establish the following helper lemma:

Lemma 11 (fwd-helper)

$$\forall (\eta, cs, ds, h) \in State \times D_c^* \times D^* \times \mathbb{N} \; . \\ (\overrightarrow{\alpha}(cs) \sqsubseteq ds \wedge \eta \in cs[h]) \Rightarrow allMeaningsHold(ds[h], \eta)$$

Proof

We pick $(\eta, cs, ds, h) \in State \times D_c^* \times D^* \times \mathbb{N}$, assume $(\overrightarrow{\alpha}(cs) \sqsubseteq ds \wedge \eta \in cs[h])$, and show $allMeaningsHold(ds[h], \eta)$.

By the definition of *allMeaningsHold*, which can be found in Definition 23, we need to show:

$$\forall (EF, (t_1, \ldots, t_i)) \in EdgeFact \times GroundTerm^i. \\ EF(t_1, \ldots, t_i) \in ds[h] \Rightarrow [\![EF]\!](t_1, \ldots, t_i, \eta)$$

To do this, pick $(EF, (t_1, \ldots, t_i)) \in EdgeFact \times GroundTerm^i$, assume $EF(t_1, \ldots, t_i) \in ds[h]$ and show:

$$[\![EF]\!](t_1, \ldots, t_i, \eta) \tag{C.7}$$

From the assumptions we know that $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, which means $\forall j \; . \; \alpha(cs[j]) \sqsubseteq ds[j]$, and using the definition of $\sqsubseteq$, $\forall j \; . \; \alpha(cs[j]) \supseteq ds[j]$. Thus, from $EF(t_1, \ldots, t_i) \in ds[h]$, we get:

$$EF(t_1, \ldots, t_i) \in \alpha(cs[h])$$

Using the definition of α (Definition 18), this means that:

$$\forall \eta \in cs[h] \; . \; [\![EF]\!](t_1, \ldots, t_i, \eta)$$

From the assumptions we know that $\eta \in cs[h]$, and thus we get $[\![EF]\!](t_1, \ldots, t_i, \eta)$, which is what we had to show in (C.7).

■

Lemma 3 *If all propagation rules in a forward Rhodium optimization O are sound then F_O as defined in* (5.3) *is sound.*

Proof

We need to show:

$$\begin{aligned} &\forall (n, cs, ds) \in Node \times D_c^* \times D^* \\ &\quad \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_O(n, ds) \end{aligned}$$

Pick $(n, cs, ds) \in Node \times D_c^* \times D^*$, assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, and show $\overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_O(n, ds)$.

To show $\overrightarrow{\alpha}(F_c(n, ds)) \sqsubseteq F_O(n, ds)$, we need to show that $\forall \; h' \; . \; \alpha(F_c(n, cs)[h']) \sqsubseteq F_O(n, ds)[h']$.

So pick h', and show $\alpha(F_c(n, cs)[h']) \sqsubseteq F_O(n, ds)[h']$, which, using the definition of $\sqsubseteq$ is $\alpha(F_c(n, cs)[h']) \supseteq F_O(n, ds)[h']$. To show this, pick $x \in F_O(n, ds)[h']$, and show that:

$$x \in \alpha(F_c(n, cs)[h']) \tag{C.8}$$

Using the definition of F_O from Equation (5.3), $x \in F_O(n, ds)[h']$ implies that there exists a θ, EF, i, $(t_1, \ldots, t_i)$, and ψ such that:

$$x = \theta(EF(t_1, \ldots, t_i)) \tag{C.9}$$

$$(\texttt{if } \psi \texttt{ then } EF(t_1, \ldots, t_i)@\texttt{out}[h']) \in O \tag{C.10}$$

$$[\![\psi]\!](\theta, ds, n) \tag{C.11}$$

Using the definition of α from (5.4), we get (where I renamed the $\forall\eta$ quantifier to $\forall\eta'$):

$$\begin{aligned} \alpha(F_c(n, cs)[h']) = \{EF(t_1, \ldots, t_j) \mid \quad & EF \in \mathit{EdgeFact}_O \wedge j = \mathit{arity}(EF) \wedge \\ & (t_1, \ldots, t_j) \in \mathit{GroundTerm}^j \wedge \\ & \forall \eta' \in F_c(n, cs)[h'] \,.\, [\![EF]\!](t_1, \ldots, t_j, \eta')\} \end{aligned} \tag{C.12}$$

To show (C.8), because of (C.9) and the fact that $\theta(EF(t_1, \ldots, t_i)) = EF(\theta(t_1), \ldots, \theta(t_i))$, we must show that $EF(\theta(t_1), \ldots, \theta(t_i)) \in \alpha(F_c(n, cs)[h']$. Using (C.12), this amounts to showing that the following two conditions hold:

$$EF \in \mathit{EdgeFact}_O \wedge i = \mathit{arity}(EF) \wedge (\theta(t_1), \ldots, \theta(t_i)) \in \mathit{GroundTerm}^i \tag{C.13}$$

$$\forall \eta' \in F_c(n, cs)[h'] \,.\, [\![EF]\!](\theta(t_1), \ldots, \theta(t_i), \eta') \tag{C.14}$$

Condition (C.13) follows directly from (C.10), and the fact that rules in O satisfy basic type correctness requirements.

To show (C.14), pick $\eta' \in F_c(n, cs)[h']$, and show:

$$[\![EF]\!](\theta(t_1), \ldots, \theta(t_i), \eta') \tag{C.15}$$

From $\eta' \in F_c(n, cs)[h']$ and the definition of F_c from Equation (5.1), we know that there exists $\eta \in \mathit{State}$ and $h \in \mathbb{N}$ such that:

$$\eta \in cs[h] \tag{C.16}$$

$$h, \eta \overset{n}{\hookrightarrow} h', \eta' \tag{C.17}$$

Because all propagation rules in the Rhodium optimization O are sound, we know from (C.10) that the rule $(\texttt{if } \psi \texttt{ then } EF(t_1, \ldots, t_i)@\texttt{out}[h'])$ satisfies (fwd-prop-sound).

We instantiate (fwd-prop-sound): the first two conditions of the antecedent are met from (C.11) and (C.17), and the third condition of the antecedent follows from (C.16), $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and Lemma 11. By instantiating (fwd-prop-sound), we get (C.15), which is what we had to show.

■

Lemma 4 *If all transformation rules in a forward Rhodium program O are sound, then R_O as defined in* (5.5) *is sound.*

Proof

We need to show:

$$\begin{array}{l} \forall (n, ds, g) \in Node \times D^* \times Graph. \\ \quad R_O(n, ds) = g \Rightarrow \\ \quad\quad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ \quad\quad\quad F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)] \end{array} \tag{C.18}$$

Pick $(n, ds, g) \in Node \times D^* \times Graph$, assume $R_O(n, ds) = g$, then pick $cs \in D_c^*$, assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and show:

$$F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

By the definition of R_O from Equation (5.5), and from $R_O(n, ds) = g$, we know that there exists ψ, θ and s such that:

$$(\texttt{if } \psi \texttt{ then transform } s) \in O \tag{C.19}$$

$$g = singleNodeGraph(n, \theta(s)) \tag{C.20}$$

$$[\![\psi]\!](\theta, ds, n) \tag{C.21}$$

$$stmtAt(n') = \theta(s) \text{ where } n' \text{ is the node from the single-node graph } g \tag{C.22}$$

From (C.20), g is a single node CFG. Furthermore, from (C.22) we know that n' is the node in g. As a result, we get:

$$\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) = F_c(n', cs)$$

Thus, we need to show:

$$F_c(n, cs) \sqsubseteq_c F_c(n', cs)$$

Or:

$$\forall\ h'\ .F_c(n, cs)[h'] \sqsubseteq_c F_c(n', cs)[h']$$

Pick an h', and show:

$$F_c(n, cs)[h'] \sqsubseteq_c F_c(n', cs)[h']$$

By the definition of $\sqsubseteq_c$, this is:

$$F_c(n, cs)[h'] \subseteq F_c(n', cs)[h']$$

To show this, pick $\eta' \in F_c(n, cs)[h']$, and show:

$$\eta' \in F_c(n', cs)[h'] \tag{C.23}$$

From $\eta' \in F_c(n, cs)[h']$ and the definition of F_c from Equation (5.1), we know that there exists $\eta \in \mathit{State}$ and $h \in \mathbb{N}$ such that:

$$\eta \in cs[h] \tag{C.24}$$

$$h, \eta \overset{n}{\hookrightarrow} h', \eta' \tag{C.25}$$

Because all transformation rules in the Rhodium optimization O are sound, we know from (C.19) that the rule `if` ψ `then transform` s satisfies (fwd-trans-sound).

We instantiate (fwd-trans-sound): the first three conditions of the antecedent are met from (C.21), (C.25) and (C.22), and the fourth condition of the antecedent follows from (C.24), $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and Lemma 11. By instantiating (fwd-trans-sound), we get $h, \eta \overset{n'}{\hookrightarrow} h', \eta'$.

Since $h, \eta \overset{n'}{\hookrightarrow} h', \eta'$, and since $\eta \in cs[h]$ from (C.24), by the definition of F_c from Equation (5.1), we get that $\eta' \in F_c(n', cs)[h']$, which is what we had to show in (C.23).

■

Appendix D

ADDITIONAL MATERIAL FOR BACKWARD RHODIUM OPTIMIZATIONS

D.1 Proofs

Lemma 5 *The backward abstraction function α from Definition 31 is join-monotonic.*

Proof

We need to show: $\alpha(\bigsqcup_c Y) \sqsubseteq \bigsqcup\{\alpha(c) \mid c \in Y\}$, where Y is any chain of the lattice domain D_c.

For our lattice, we will prove the following stronger property:

$$\forall Y \in 2^{D_c} \;.\; \alpha(\bigsqcup{}_c Y) \sqsubseteq \bigsqcup\{\alpha(c) \mid c \in Y\} \tag{D.1}$$

This property is stronger because it holds for any set Y of D_c elements, not only for chains.

To show (D.1), pick $Y \in 2^{D_c}$, and prove:

$$\alpha(\bigsqcup{}_c Y) \sqsubseteq \bigsqcup\{\alpha(c) \mid c \in Y\}$$

Using the definition of $\sqcup_c$ and $\sqcup$, this becomes:

$$\alpha(\bigcup Y) \supseteq \bigcap\{\alpha(c) \mid c \in Y\}$$

Let $Q = \bigcup Y$ and let $R = \bigcap\{\alpha(c) \mid c \in Y\}$. Note that because the type of α is $\alpha : D_c \to D$, we have that R is a set of elements of D, and therefore $R \subseteq D$.

With these definitions, we now need to show $\alpha(Q) \supseteq R$. To do this, assume $f \in R$ and show $f \in \alpha(Q)$. From $R \subseteq D$, $f \in R$, and the definition of D (Definition 14), we get:

$$f = EF(t_1, \ldots, t_i) \tag{D.2}$$

$$EF \in EdgeFact_O \wedge i = arity(EF) \wedge (t_1, \ldots, t_i) \in GroundTerm^i \tag{D.3}$$

Our assumption is therefore $EF(t_1, \ldots, t_i) \in R$, and we are trying to show $EF(t_1, \ldots, t_i) \in \alpha(Q)$. To do this, by the definition of α (Definition 31), we need to show:

$$EF \in EdgeFact_O \wedge i = arity(EF) \wedge (t_1, \ldots, t_i) \in GroundTerm^i \tag{D.4}$$

$$[\![EF]\!](t_1, \ldots, t_i, Q) \tag{D.5}$$

Condition (D.4) is already provided from (D.3). So we are only left with showing (D.5).

From the assumption introduced to show $\alpha(Q) \supseteq R$, we have $EF(t_1, \ldots, t_i) \in R$, and since $R = \bigcap\{\alpha(c) \mid c \in Y\}$, we get from the definition of $\bigcap$:

$$\forall c \in Y \;.\; EF(t_1, \ldots, t_i) \in \alpha(c)$$

which, from the definition of α (Definition 31), gives us:

$$\forall c \in Y \;.\; [\![EF]\!](t_1, \ldots, t_i, c) \tag{D.6}$$

Now we can show (D.5). There are two cases:

- EF is a predicate-fact schema. In this case, from the definition of $[\![EF]\!]$ for predicate-fact schemas (Definition 32), Equation (D.6) becomes:

  $$\forall c \in Y \;.\; \forall \eta \in c \;.\; [\![EF]\!]_p(t_1, \ldots, t_i, \eta) \tag{D.7}$$

  Also, from the definition of $[\![EF]\!]$ for predicate-fact schemas (Definition 32), Equation (D.5), which we need to show, becomes:

  $$\forall \eta \in Q \;.\; [\![EF]\!]_p(t_1, \ldots, t_i, \eta)$$

  To show this, pick $\eta \in Q$, and show $[\![EF]\!]_p(t_1, \ldots, t_i, \eta)$.

  Since $Q = \bigcup Y$, from $\eta \in Q$ and the definition of $\bigcup$, we get that there exists a c such that $c \in Y \wedge \eta \in c$. Instantiating (D.7) with c and η, we get $[\![EF]\!]_p(t_1, \ldots, t_i, \eta)$, which is what we had to show.

- *EF* is a relational-fact schema. In this case, from the definition of $[\![EF]\!]$ for relational-fact schemas (Definition 33), Equation (D.6) becomes:

$$\forall c \in Y \;.\; \forall(\eta_1, \eta_2) \in State^2 \;.\; [\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (\eta_1 \in c \Leftrightarrow \eta_2 \in c) \tag{D.8}$$

  Also, from the definition of $[\![EF]\!]$ for relational-fact schemas (Definition 33), Equation (D.5), which we need to show, becomes:

$$\forall(\eta_1, \eta_2) \in State^2 \;.\; [\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (\eta_1 \in Q \Leftrightarrow \eta_2 \in Q)$$

  To show this, pick $(\eta_1, \eta_2) \in State^2$, assume $[\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2)$, and show $\eta_1 \in Q \Leftrightarrow \eta_2 \in Q$. To do this, we show $\eta_1 \in Q \Rightarrow \eta_2 \in Q$ and then $\eta_2 \in Q \Rightarrow \eta_1 \in Q$.

  – To show $\eta_1 \in Q \Rightarrow \eta_2 \in Q$, assume $\eta_1 \in Q$, and show $\eta_2 \in Q$.

    Since $Q = \bigcup Y$, from $\eta_1 \in Q$ and the definition of $\bigcup$, we get that there exists a c such that $c \in Y \wedge \eta_1 \in c$. Instantiating (D.8) with c, η_1, and η_2, we get:

$$[\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (\eta_1 \in c \Leftrightarrow \eta_2 \in c) \tag{D.9}$$

    We've already assumed $[\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2)$, and we know $\eta_1 \in c$, so (D.9) gives us $\eta_2 \in c$. Since $c \in Y$ and $Q = \bigcup Y$, by the definition of $\bigcup$, $\eta_2 \in c$ implies $\eta_2 \in Q$, which is what we had to show.

  – To show $\eta_2 \in Q \Rightarrow \eta_1 \in Q$, we proceed in a way that is exactly symmetric to the $\eta_1 \in Q \Rightarrow \eta_2 \in Q$ case.

■

Before proving Lemmas 6 and 7, I first establish the following two helper lemmas:

Lemma 12 (bwd-helper-1)

$$\begin{aligned}&\forall(\eta, cs, ds, h) \in State \times D_c^* \times D^* \times \mathbb{N} \;. \\ &\quad (\overrightarrow{\alpha}(cs) \sqsubseteq ds \wedge \eta \in cs[h]) \Rightarrow allMeaningsHold_p(ds[h], \eta)\end{aligned}$$

Proof

We pick $(\eta, cs, ds, h) \in State \times D_c^* \times D^* \times \mathbb{N}$, assume $(\overrightarrow{\alpha}(cs) \sqsubseteq ds \wedge \eta \in cs[h])$, and show $allMeaningsHold_p(ds[h], \eta)$.

By the definition of $allMeaningsHold_p$, which can be found in Definition 36, we need to show:

$$\forall (EF, (t_1, \ldots, t_i)) \in EdgeFact_p \times GroundTerm^i.$$
$$EF(t_1, \ldots, t_i) \in ds[h] \Rightarrow [\![EF]\!]_p(t_1, \ldots, t_i, \eta)$$

To do this, pick $(EF, (t_1, \ldots, t_i)) \in EdgeFact_p \times GroundTerm^i$, assume $EF(t_1, \ldots, t_i) \in ds[h]$ and show:

$$[\![EF]\!]_p(t_1, \ldots, t_i, \eta) \tag{D.10}$$

From the assumptions we know that $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, which means $\forall j \; . \; \alpha(cs[j]) \sqsubseteq ds[j]$, and using the definition of $\sqsubseteq$, $\forall j \; . \; \alpha(cs[j]) \supseteq ds[j]$. Thus, from $EF(t_1, \ldots, t_i) \in ds[h]$, we get:

$$EF(t_1, \ldots, t_i) \in \alpha(cs[h])$$

Using the definition of α (Definition 31), this means that:

$$[\![EF]\!](t_1, \ldots, t_i, cs[h])$$

which, because EF is a backward-predicate-fact schema, means:

$$\forall \eta \in cs[h] \; . \; [\![EF]\!]_p(t_1, \ldots, t_i, \eta)$$

From the assumptions we know that $\eta \in cs[h]$, and thus we get $[\![EF]\!]_p(t_1, \ldots, t_i, \eta)$, which is what we had to show in (D.10).

■

Lemma 13 (bwd-helper-2)

$$\forall (\eta_1, \eta_2, cs, ds, h) \in State \times State \times D_c^* \times D^* \times \mathbb{N} \; .$$
$$(\overrightarrow{\alpha}(cs) \sqsubseteq ds \wedge someMeaningHolds(ds[h], \eta_1, \eta_2)) \Rightarrow (\eta_1 \in cs[h] \Leftrightarrow \eta_2 \in cs[h])$$

Proof

We pick $(\eta_1, \eta_2, cs, ds, h) \in \mathit{State} \times \mathit{State} \times D_c^* \times D^* \times \mathbb{N}$, we assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and $\mathit{someMeaningHolds}(ds[h], \eta_1, \eta_2)$, and we show:

$$\eta_1 \in cs[h] \Leftrightarrow \eta_2 \in cs[h]$$

From $\mathit{someMeaningHolds}(ds[h], \eta_1, \eta_2)$ and the definition of *someMeaningHolds* (which can be found in Definition 37), we know that there exists $(EF, (t_1, \ldots, t_i)) \in \mathit{EdgeFact}_r \times \mathit{GroundTerm}^i$ such that:

$$EF(t_1, \ldots, t_i) \in ds[h] \tag{D.11}$$

$$[\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \tag{D.12}$$

We have:

$\overrightarrow{\alpha}(cs) \sqsubseteq ds$

$\Rightarrow$ $\alpha(cs[h]) \sqsubseteq ds[h]$

$\Rightarrow$ $\alpha(cs[h]) \supseteq ds[h]$ (using the definition of $\sqsubseteq$)

$\Rightarrow$ $EF(t_1, \ldots, t_i) \in \alpha(cs[h])$ (using (D.11))

$\Rightarrow$ $[\![EF]\!](t_1, \ldots, t_i, cs[h])$ (using the definition of α from Equation (6.3))

$\Rightarrow$ $\forall(\eta_1, \eta_2) \in \mathit{State}^2 \,.[\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (\eta_1 \in cs[h] \Leftrightarrow \eta_2 \in cs[h])$

(using Definition 33, which defines $[\![EF]\!]$ for backward-relational-fact schemas)

$\Rightarrow$ $[\![EF]\!]_r(t_1, \ldots, t_i, \eta_1, \eta_2) \Rightarrow (\eta_1 \in cs[h] \Leftrightarrow \eta_2 \in cs[h])$

(instantiating with η_1 and η_2)

$\Rightarrow$ $\eta_1 \in cs[h] \Leftrightarrow \eta_2 \in cs[h]$ (using (D.12))

■

Lemma 6 *If all propagation rules in a backward Rhodium optimization O are sound then F_O as defined in* (6.2) *is sound.*

Proof

We need to show:

$$\forall (n, cs, ds) \in \mathit{Node} \times D_c^* \times D^* \\ \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_O(n, ds)$$

Pick $(n, cs, ds) \in \mathit{Node} \times D_c^* \times D^*$, assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, and show $\overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_O(n, ds)$.

To show $\overrightarrow{\alpha}(F_c(n, ds)) \sqsubseteq F_O(n, ds)$, we need to show that $\forall\ h'\ .\ \alpha(F_c(n, cs)[h']) \sqsubseteq F_O(n, ds)[h']$.

So pick h', and show $\alpha(F_c(n, cs)[h']) \sqsubseteq F_O(n, ds)[h']$, which, using the definition of $\sqsubseteq$ is $\alpha(F_c(n, cs)[h']) \supseteq F_O(n, ds)[h']$. To show this, pick $x \in F_O(n, ds)[h']$, and show that:

$$x \in \alpha(F_c(n, cs)[h']) \tag{D.13}$$

Using the definition of F_O from Equation (6.2), $x \in F_O(n, ds)[h']$ implies that there exists a θ, EF, i, $(t_1, \ldots, t_i)$, and ψ such that:

$$x = \theta(EF(t_1, \ldots, t_i)) \tag{D.14}$$

$$(\texttt{if}\ \psi\ \texttt{then}\ EF(t_1, \ldots, t_i)@\texttt{in}[h']) \in O \tag{D.15}$$

$$[\![\psi]\!]_b(\theta, ds, n) \tag{D.16}$$

Using the definition of α from (6.3), we get:

$$\begin{aligned} \alpha(F_c(n, cs)[h']) = \{EF(t_1, \ldots, t_j) \mid\ & EF \in \mathit{EdgeFact}_O \wedge j = \mathit{arity}(EF) \wedge \\ & (t_1, \ldots, t_j) \in \mathit{GroundTerm}^j \wedge \\ & [\![EF]\!](t_1, \ldots, t_j, F_c(n, cs)[h']) \end{aligned} \tag{D.17}$$

To show (D.13), because of (D.14) and the fact that $\theta(EF(t_1, \ldots, t_i)) = EF(\theta(t_1), \ldots, \theta(t_i))$, we must show that $EF(\theta(t_1), \ldots, \theta(t_i)) \in \alpha(F_c(n, cs)[h']$. Using (D.17), this amounts to showing that the following two conditions hold:

$$EF \in \mathit{EdgeFact}_O \wedge i = \mathit{arity}(EF) \wedge (\theta(t_1), \ldots, \theta(t_i)) \in \mathit{GroundTerm}^i \tag{D.18}$$

$$[\![EF]\!](\theta(t_1), \ldots, \theta(t_i), F_c(n, cs)[h']) \tag{D.19}$$

Condition (D.18) follows directly from (D.15), and the fact that rules in O satisfy basic type correctness requirements.

To show (D.19), there are two cases:

- *EF* is a predicate-fact schema. Then, by the definition of $[\![EF]\!]$ for predicate-fact schemas (Definition 32), the condition that we need to show, condition (D.19), becomes:

$$\forall \eta' \in F_c(n, cs)[h'] \; . \; [\![EF]\!]_p(\theta(t_1), \ldots, \theta(t_i), \eta')$$

  To show this, pick $\eta' \in F_c(n, cs)[h']$, and show:

$$[\![EF]\!]_p(\theta(t_1), \ldots, \theta(t_i), \eta') \tag{D.20}$$

  From $\eta' \in F_c(n, cs)[h']$ and the definition of F_c from Equation (6.1), we know that there exists $\eta \in \mathit{State}$ and $h \in \mathbb{N}$ such that:

$$\eta \in cs[h] \tag{D.21}$$

$$h', \eta' \overset{n}{\hookrightarrow} h, \eta \tag{D.22}$$

  Because all propagation rules in the Rhodium optimization O are sound, we know from (D.15) that the rule (`if` ψ `then` $EF(t_1, \ldots, t_i)$`@out`$[h']$) must be sound. Since *EF* is predicate-fact schema, this means that the rule satisfies (bwd-prop-sound-pr).

  We instantiate (bwd-prop-sound-pr): the first two conditions of the antecedent are met from (D.16) and (D.22), and the third condition of the antecedent follows from (D.21), $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and Lemma 12. By instantiating (bwd-prop-sound-pr), we get (D.20), which is what we had to show.

- *EF* is a relational-fact schema. Then, by the definition of $[\![EF]\!]$ for relational-fact schemas (Definition 33), the condition that we need to show, condition (D.19), becomes (where I have renamed the η_1, η_2 quantifier to η_1', η_2'):

$$\forall (\eta_1', \eta_2') \in \mathit{State}^2 \; .$$
$$[\![EF]\!]_r(\theta(t_1), \ldots, \theta(t_i), \eta_1', \eta_2') \Rightarrow (\eta_1' \in F_c(n, cs)[h'] \Leftrightarrow \eta_2' \in F_c(n, cs)[h'])$$

To show this, pick $(\eta_1', \eta_2') \in State^2$, assume:

$$[\![EF]\!]_r(\theta(t_1), \ldots, \theta(t_i), \eta_1', \eta_2') \tag{D.23}$$

and show:

$$(\eta_1' \in F_c(n, cs)[h'] \Leftrightarrow \eta_2' \in F_c(n, cs)[h'])$$

To show this, we show $\eta_1' \in F_c(n, cs)[h'] \Rightarrow \eta_2' \in F_c(n, cs)[h']$, and then $\eta_2' \in F_c(n, cs)[h'] \Rightarrow \eta_1' \in F_c(n, cs)[h']$.

– To show $\eta_1' \in F_c(n, cs)[h'] \Rightarrow \eta_2' \in F_c(n, cs)[h']$, we assume $\eta_1' \in F_c(n, cs)[h']$, and show $\eta_2' \in F_c(n, cs)[h']$.

From the definition of F_c in Equation (6.1), we get (where I have renamed η, η' and i to η', η and h):

$$F_c(n, cs)[h'] = \{\eta' \mid \exists\ \eta \in State, h \in \mathbb{N}\ .\ [\eta \in cs[h]\ \wedge\ h', \eta' \overset{n}{\hookrightarrow} h, \eta]\} \tag{D.24}$$

Since $\eta_1' \in F_c(n, cs)[h']$, we therefore have:

$$\exists\ \eta \in State, h \in \mathbb{N}\ .\ [\eta \in cs[h]\ \wedge\ h', \eta_1' \overset{n}{\hookrightarrow} h, \eta]$$

Skolemizing this existential with η_1 for η and h for h, we get:

$$\eta_1 \in cs[h] \tag{D.25}$$

$$h', \eta_1' \overset{n}{\hookrightarrow} h, \eta_1 \tag{D.26}$$

Because all propagation rules in the Rhodium optimization O are sound, we know from (D.15) that the rule (`if` ψ `then` $EF(t_1, \ldots, t_i)$`@out`$[h']$) must be sound. Since EF is relational-fact schema, this means that the rule satisfies (bwd-prop-sound-rel-1) and (bwd-prop-sound-rel-2).

We instantiate (bwd-prop-sound-rel-1): the first condition of the antecedent is met from (D.23), and the second condition is met from (D.16). By instantiating (bwd-prop-sound-rel-1), we get (where I have renamed the η_1' and η_2' quantifiers to η_1 and η_2):

$$\exists \eta_1 \in State\ .\ h', \eta_1' \overset{n}{\hookrightarrow} h, \eta_1\ \Leftrightarrow \exists \eta_2 \in State\ .\ h', \eta_2' \overset{n}{\hookrightarrow} h, \eta_2$$

Combined with (D.26), this gives $\exists \eta_2 \in \mathit{State}\ .\ h', \eta_2' \overset{n}{\hookrightarrow} h, \eta_2$, which skolemized, gives us:

$$h', \eta_2' \overset{n}{\hookrightarrow} h, \eta_2 \tag{D.27}$$

To show $\eta_2' \in F_c(n, cs)[h']$, using Equation (D.24) and (D.27), it suffices to show:

$$\eta_2 \in cs[h] \tag{D.28}$$

We now instantiate (bwd-prop-sound-rel-2): the first condition of the antecedent is met from (D.23), the second is met from (D.16), the third is met from (D.26), and the fourth is met from (D.27). By instantiating (bwd-prop-sound-rel-2), we get:

$$\eta_1 = \eta_2 \vee \mathit{someMeaningHolds}(ds[h], \eta_1, \eta_2)$$

If $\eta_1 = \eta_2$, then from (D.25), we get $\eta_2 \in cs[h]$, which is what we had to show in (D.28). Otherwise, we have that $\mathit{someMeaningHolds}(ds[h], \eta_1, \eta_2)$. Then, from $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, $\mathit{someMeaningHolds}(ds[h], \eta_1, \eta_2)$, and Lemma 13, we get that $\eta_1 \in cs[h] \Leftrightarrow \eta_2 \in cs[h]$, which, combined with (D.25), gives $\eta_2 \in cs[h]$, which is what we had to show in (D.28).

– To show $\eta_2' \in F_c(n, cs)[h'] \Rightarrow \eta_1' \in F_c(n, cs)[h']$, we proceed in a way that is exactly symmetric to the $\eta_1' \in F_c(n, cs)[h'] \Rightarrow \eta_2' \in F_c(n, cs)[h']$ case.

■

Lemma 7 *If all transformation rules in a backward Rhodium program O are sound, then R_O as defined in* (6.4) *is sound.*

Proof

We need to show:

$$\begin{array}{l} \forall (n, ds, g) \in \mathit{Node} \times D^* \times \mathit{Graph}. \\ \quad R_O(n, ds) = g \Rightarrow \\ \quad\quad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ \quad\quad\quad F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(\mathit{OutEdges}_g)] \end{array} \tag{D.29}$$

Pick $(n, ds, g) \in Node \times D^* \times Graph$, assume $R_O(n, ds) = g$, then pick $cs \in D_c^*$, assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and show:

$$F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

By the definition of R_O from Equation (6.4), and from $R_O(n, ds) = g$, we know that there exists ψ, θ and s such that:

$$(\texttt{if } \psi \texttt{ then transform } s) \in O \tag{D.30}$$

$$g = singleNodeGraph(n, \theta(s)) \tag{D.31}$$

$$[\![\psi]\!]_b(\theta, ds, n) \tag{D.32}$$

$$stmtAt(n') = \theta(s) \text{ where } n' \text{ is the node from the single-node graph } g \tag{D.33}$$

From (D.31), g is a single node CFG. Furthermore, from (D.33) we know that n' is the node in g. As a result, we get:

$$\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) = F_c(n', cs)$$

Thus, we need to show:

$$F_c(n, cs) \sqsubseteq_c F_c(n', cs)$$

Or:

$$\forall\ h'\ .F_c(n, cs)[h'] \sqsubseteq_c F_c(n', cs)[h']$$

Pick an h', and show:

$$F_c(n, cs)[h'] \sqsubseteq_c F_c(n', cs)[h']$$

By the definition of $\sqsubseteq_c$, this is:

$$F_c(n, cs)[h'] \subseteq F_c(n', cs)[h']$$

To show this, pick $\eta' \in F_c(n, cs)[h']$, and show:

$$\eta' \in F_c(n', cs)[h'] \tag{D.34}$$

From $\eta' \in F_c(n, cs)[h']$ and the definition of F_c from Equation (6.1), we know that there exists $\eta \in \mathit{State}$ and $h \in \mathbb{N}$ such that:

$$\eta \in cs[h] \tag{D.35}$$

$$h', \eta' \overset{n}{\hookrightarrow} h, \eta \tag{D.36}$$

Because all transformation rules in the Rhodium optimization O are sound, we know from (D.30) that the rule `if` ψ `then transform` s satisfies (bwd-trans-sound).

We instantiate (bwd-trans-sound): the first three conditions of the antecedent are met from (D.32), (D.36) and (D.33), and the fourth condition of the antecedent follows from (D.35), $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and Lemma 12. By instantiating (bwd-trans-sound), we get that there exists an $\eta'' \in \mathit{State}$ such that:

$$h', \eta' \overset{n'}{\hookrightarrow} h, \eta'' \tag{D.37}$$

$$\eta = \eta'' \vee \mathit{someMeaningHolds}(ds[h], \eta, \eta'') \tag{D.38}$$

We need to show (D.34), which is $\eta' \in F_c(n', cs)[h']$. From $h', \eta' \overset{n'}{\hookrightarrow} h, \eta''$ in Equation (D.37), and from the definition of F_c in Equation (6.1), it suffices to show:

$$\eta'' \in cs[h] \tag{D.39}$$

If $\eta = \eta''$, then from (D.35), we immediately get (D.39).

Otherwise, from (D.38), we get $\mathit{someMeaningHolds}(ds[h], \eta, \eta'')$. Then, from $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, $\mathit{someMeaningHolds}(ds[h], \eta, \eta'')$, and Lemma 13, we get that $\eta \in cs[h] \Leftrightarrow \eta'' \in cs[h]$, which combined with (D.35), gives us $\eta'' \in cs[h]$, which is what had to be shown in (D.39).

■

VITA

Sorin Lerner received a B. Eng. degree in Computer Engineering from McGill University, Montreal in 1999. He received an M.S. in 2001, and a Ph.D. in 2006, both from the University of Washington in computer science. As a graduate student at the University of Washington, he worked with professor Craig Chambers on trustworthy compilation frameworks. During internships at Microsoft Research, he also worked on dynamic optimization techniques for x86 code, and designs of scalable analyses for large C/C++ applications.