Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules

Sorin Lerner
Univ. of Washington
lerns@cs.washington.edu

Todd Millstein
UCLA
todd@cs.ucla.edu

Erika Rice
Univ. of Washington
erice@cs.washington.edu

Craig Chambers
Univ. of Washington
chambers@cs.washington.edu

ABSTRACT

We present Rhodium, a new language for writing compiler optimizations that can be automatically proved sound. Unlike our previous work on Cobalt, Rhodium expresses optimizations using explicit dataflow facts manipulated by local propagation and transformation rules. This new style allows Rhodium optimizations to be mutually recursively defined, to be automatically composed, to be interpreted in both flow-sensitive and -insensitive ways, and to be applied interprocedurally given a separate context-sensitivity strategy, all while retaining soundness. Rhodium also supports infinite analysis domains while guaranteeing termination of analysis. We have implemented a soundness checker for Rhodium and have specified and automatically proven the soundness of all of Cobalt's optimizations plus a variety of optimizations not expressible in Cobalt, including Andersen's points-to analysis, arithmetic-invariant detection, loop-induction-variable strength reduction, and redundant array load elimination.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification – *correctness proofs, reliability, validation*; D.3.4 [Programming Languages]: Processors – *compilers, optimization*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs – *mechanical verification*

General Terms: Reliability, languages, verification.

Keywords: Compiler optimization, automated correctness proofs.

1. INTRODUCTION

Compilers are an important part of a programmer's computing infrastructure. If the compiler doesn't generate correct code, the whole application being compiled is compromised. As a result, much work has been directed toward making compilers trustworthy, including testing, translation validation [25, 23], credible compilation [26], and manual proof techniques [8, 9, 37, 14, 17, 10]. In previous work [19], we presented a system in which optimizations could be checked for soundness automatically. An optimization is sound if it is guaranteed to preserve the semantics of any program it optimizes. Our solution was centered on a domain-specific language for writing optimizations, called Cobalt. An optimization written in Cobalt was checked for soundness by asking an automatic theorem prover to discharge a small set of simple proof obligations. We proved by hand, once and for all, that if a Cobalt optimization satisfies these obligations, then the optimization is indeed sound. Unlike testing, credible compilation, and translation validation, this checking is done once when the compiler is developed, separately from any particular programs being optimized. Cobalt thus enables a key component of modern optimizing compilers to become trusted, and it opens the door for users to extend their compilers with application-specific optimizations without compromising the correctness of the compiler.

Cobalt is expressive enough to allow a range of flow-sensitive intraprocedural optimizations to be defined and proved correct. Using Cobalt, we were able to write and automatically check the soundness of constant propagation, copy propagation, dead-assignment elimination, common subexpression elimination, partial redundancy elimination, partial dead-code elimination, and simple kinds of pointer analyses. However, Cobalt's design, where optimizing transformations are triggered based on a restricted temporal-logic predicate over the entire control flow graph (CFG), imposes limits that make it difficult to extend to a wider range of optimizations.

In this paper we present Rhodium, a new domain-specific language for optimizations that can express a much greater range of optimizations while still proving them sound automatically. The key technical change from Cobalt is to make dataflow facts explicit (rather than implicit in a temporal-logic predicate) and to use a separate and extensible set of *local* propagation and transformation rules to generate new dataflow facts from old dataflow facts and to specify when statements are optimized based on inflowing dataflow facts. Each dataflow fact is given a semantic meaning, in the form of a predicate over program states. To prove a Rhodium optimization correct, our system asks an automatic theorem prover to discharge a local soundness lemma for each propagation and transformation rule, using the meanings of the facts manipulated by the rules and the concrete semantics of the program's statements. We proved, once by hand, that these lemmas imply that the optimization is globally sound. Because Rhodium's local propagation model is fundamentally different from Cobalt's, this hand proof is also fundamentally different, and couched in terms of abstract interpretation [8].

Rhodium's use of explicit dataflow facts with local propagation and transformation rules enables several important advances over Cobalt's use of global temporal-logic predicates:

- **Traditional form.** A local propagation rule is a kind of flow- or transfer function, which may be a more comfortable and understandable model for an optimization writer than Cobalt's global model.
- **Extensibility.** Rhodium allows new propagation rules to be added without modifying any existing rules or fact definitions, enabling optimizations to be enhanced more easily.
- **Recursively defined analyses.** When deciding whether to generate a particular dataflow fact on a statement's successor edge, a Rhodium propagation rule can examine any other dataflow facts on the statement's predecessor edges. Cobalt was in effect only able to propagate the same dataflow fact through a statement unchanged. Rhodium allows the propagation rules of dataflow facts to be defined mutually recursively, significantly increasing their expressiveness and clarity.
- **Composed analyses and transformations.** By using a model based on local propagation and transformation rules, we can exploit previous work on automatically composing analyses and transformations [18] to enable Rhodium optimizations to be automatically composed.
- **Flow-insensitive analyses.** We show how to interpret Rhodium propagation rules in a flow-insensitive manner, soundly, yielding more-efficient analyses with no extra optimization-writer work. In contrast, Cobalt's global model was inherently flow-sensitive.
- **Interprocedural analyses.** We show how to define a context-sensitive interprocedural analysis from a Rhodium intraprocedural analysis and a specification of a context-sensitivity strategy. Rhodium's local propagation model allows the local propagation rule for call statements to be derived automatically. If the intraprocedural analysis is sound, then the interprocedural one is sound, too.

In addition to moving to a local propagation model, we have also enriched Rhodium's expressiveness in the following orthogonal ways:

- **Dynamic semantics extensions.** Rhodium allows the optimization-writer to define "virtual" extensions to the intermediate language's dynamic semantics which can compute properties of program execution traces. For example, the statement at which each memory location was allocated can be computed via a dynamic semantics extension. These extensions can then be referenced in the meanings of dataflow facts, for instance in a points-to analysis with allocation-site heap summaries, enabling a wider class of optimizations to be proved sound automatically.
- **Infinite analysis domains.** Rhodium allows dataflow-fact domains to be infinite, leading to increased expressiveness over Cobalt which only allowed finite domains (such as the set of constants, variables, and expressions that appeared in the program being optimized). We present sufficient conditions, including some adapted from the database community, for automatically guaranteeing that analyses terminate even in the face of such infinite domains. Rhodium analyses can also specify widening operators [8], without affecting soundness.

The end result is a language that is significantly more expressive than Cobalt but nonetheless provides the same strong soundness guarantees. We have implemented our strategy for automatically proving Rhodium analyses and optimizations sound using Simplify, the automatic theorem prover from ESC/Java [12]. We defined and automatically proved sound all of Cobalt's optimizations plus the following new optimizations and analyses that were not expressible in Cobalt: loop-induction-variable strength reduction, a flow-sensitive version of Andersen's points-to analysis [3] with heap summaries, arithmetic invariant detection, constant propagation through array elements, redundant array load elimination, and integer range analysis. Our Rhodium code defines 24 dataflow facts, 105 propagation rules, and 14 transformation rules. Moreover, all these analyses can be interpreted as flow-insensitive analyses and/or context-sensitive or -insensitive interprocedural analyses, and they can be automatically composed together to yield more-precise solutions, soundly.

Section 2 introduces the new flow-function-oriented way of writing optimizations in Rhodium and describes the associated automated proof strategy based on abstract interpretation. Section 3 presents our technique for reducing the complexity of proof obligations using extensions to the dynamic semantics and shows how our technique can be used to reason automatically about heap summaries. Section 4 describes how we support infinite analysis domains while still being able to guarantee termination. Sections 5 and 6 present our frameworks for building provably sound flow-insensitive and interprocedural optimizations. Section 7 discusses our execution engine for Rhodium in the Whirlwind compiler. Finally, sections 8 and 9 discuss future work and related work, respectively.

2. 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. This section describes how intraprocedural, flow-sensitive analyses are expressed and automatically proven sound in Rhodium; sections 5 and 6 respectively discuss flow-insensitive and interprocedural analyses. Rhodium optimizations operate over a CFG representation of the IL program, with each node representing a simple register-transfer-level statement.

Dataflow information is encoded in Rhodium by means of dataflow facts, which are user-defined function symbols applied to a set of terms, for example $\mathit{hasConstValue}(\mathtt{x}, 5)$ or $\mathit{exprIsAvailable}(\mathtt{x}, \mathtt{a} + \mathtt{b})$. A Rhodium analysis uses *propagation rules*, which are a stylized way of writing flow functions, to specify how dataflow facts propagate across CFG

```
1. decl X: Var, Y: Var, Z: Var

2. define edge fact mustNotPointTo(X: Var, Y: Var)
3. with meaning $\sigma(X) \neq \sigma(\&Y)$

4. if stmt($X := \&Z$) $\wedge\ Y \neq Z$
5. then mustNotPointTo$(X, Y)$@out

6. if mustNotPointTo$(X, Y)$@in $\wedge$ mustNotDef$(X)$
7. then mustNotPointTo$(X, Y)$@out
```

Figure 1: Simple pointer analysis in Rhodium.

nodes. These user-defined flow functions implicitly define a dataflow analysis, whose solution is the least fixed point of the standard equations induced by the flow functions. Once an analysis has reached a fixed point, the computed information can be used by Rhodium *transformation rules* to rewrite some of the CFG's nodes.

We wish to automatically prove Rhodium analyses and transformations *sound*. An analysis is sound if, for all IL procedures P, the dataflow information computed for P is consistent with the procedure's concrete semantics. A transformation is sound if, for all IL procedures P, the transformation preserves P's semantics.

Section 2.1 illustrates Rhodium's propagation rules, and section 2.2 describes how such rules are automatically proven sound using abstract interpretation. In section 2.3 we compare Rhodium's design and proof strategy with those of Cobalt and show the expressiveness benefits of our new design in Rhodium. Section 2.4 discusses Rhodium transformations and how they are automatically proven sound. Section 2.5 shows how to incorporate profitability information into Rhodium optimizations.

2.1 Propagation Rules

We illustrate Rhodium's propagation rules with a simple pointer analysis, shown in figure 1. The analysis determines that a variable `x` definitely does not point to another variable `y` if `x` was assigned the address of a variable different from `y`, and then `x` was not modified afterwards. Because our strategy for automated soundness checking is geared toward *must* analyses, we encode our pointer information using the must-not-point-to relation instead of the may-point-to relation. Each edge in the CFG will therefore be annotated with facts of the form *mustNotPointTo*(X, Y), where X and Y range over variables in the associated IL procedure. The declaration of the *mustNotPointTo edge fact* is shown on line 2 of the Rhodium code (for now the meaning on line 3 can be ignored).

Propagation rules in Rhodium indicate how edge facts are propagated across CFG nodes. For example, the rule on lines 6-7 of figure 1 defines a condition for preserving a *mustNotPointTo* fact across a node: 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 left-hand side of a rule is called the *antecedent* and the right-hand side the *consequent*. Each propagation rule is interpreted within the context of a CFG node. Edge facts are followed by @ signs, with the name after the @ sign indicating the edge on which the fact appears. For example, *mustNotPointTo*(X, Y)@*in* is true if the incoming CFG edge of the current node is annotated with *mustNotPointTo*(X, Y). Facts without @ signs are *node facts*, and they represent information about the current node. For example, the user-defined *mustNotDef*(X) fact holds at a node if the node does not modify X. An accompanying technical report [20] shows how users can define these node facts.

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, the *mustNotPointTo*(X, Y) fact will be propagated on the outgoing edge of a node for each substitution of X and Y with variables that makes the antecedent valid.

While the rule in lines 6-7 of figure 1 specifies how to preserve *mustNotPointTo* facts, the rule in lines 4-5 specifies how to introduce them in the first place. That rule 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.

All rules in figure 1 are *forward*: the antecedent only refers to a node's incoming CFG edge and the consequent only refers to a node's outgoing CFG edge. Rhodium also supports backward rules, where the antecedent only refers to *out* and the consequent only refers to *in*. The primary focus of our Rhodium work so far has been on forward analyses and transformations, and so we do not present any backward rules here. Also, for brevity and clarity, we only present definitions and theorems for the forward case, with the backward case covered in the accompanying technical report [20]. Section 8 discusses the state of backward analyses and transformations in Rhodium.

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 dataflow facts. Each edge in the CFG is therefore annotated with a set of dataflow facts, where bigger sets are more precise than smaller sets.[1] 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.

Formally, the flow function F is defined in terms of the meaning of an antecedent ψ, which is given by the function $[\![\psi]\!] : Node \times D \times Subst \to bool$ (where *Node* is the set of all CFG nodes and *Subst* is the set of all substitutions). Given a node n, a set of facts d, and a substitution θ, $[\![\psi]\!](n, d, \theta)$ is true iff $\theta(\psi)$ holds at node n with incoming facts d (where $\theta(\cdot)$ represents substitution application). The definition of $[\![\psi]\!]$ is straightforward, with the interesting case being:

$$[\![f(\overrightarrow{t})@in]\!](n, d, \theta) = f(\theta(\overrightarrow{t})) \in d$$
$$\text{(where } \overrightarrow{t} \text{ denotes a sequence of terms)}$$

A complete definition of $[\![\psi]\!]$ is given in the accompanying technical report [20]. The flow function $F : Node \times D \to D$ induced by a set R of forward propagation rules is then:

$$F(n, d) = \{\theta(f(\overrightarrow{t})) \mid [\textbf{\textit{if}}\ \psi\ \textbf{\textit{then}}\ f(\overrightarrow{t})@out] \in R \wedge [\![\psi]\!](n, d, \theta)\}$$

The solution of the induced analysis $\mathcal{A}$ is the least fixed

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

point of the standard set of dataflow equations generated from F. Although the two rules in figure 1 propagate the same dataflow fact, different rules can propagate different dataflow facts, and the fixed point is computed over all dataflow facts simultaneously.

In addition to edge facts and node facts, Rhodium also provides *virtual* dataflow facts, which can be used to define shorthands for boolean combinations of other facts. This facility allows a may-point-to fact to be defined and referred to in analyses and transformations if desired: $mayPointTo(X, Y) \triangleq \neg mustNotPointTo(X, Y)$. Such virtual facts get replaced with the boolean expression they stand for as a preprocessing step.

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 through conjunctions, disjunctions and quantifiers, we require all negation on edge facts to cancel out. The absence of negated edge facts guarantees the monotonicity of F, as shown in the accompanying technical report [20]. 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, we never found the need to use any negated edge facts, except as a notational convenience. For example, in our 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 $mustNotPointTo(X, Y)$.

2.2 Proving soundness automatically

Our goal is to ensure automatically that the dataflow information computed by the analysis $\mathcal{A}$ is sound with respect to the concrete collecting semantics of the IL. Our automatic proof strategy separates the proof that $\mathcal{A}$ is sound into two parts: the first part is analysis *dependent* and it is discharged by an automatic theorem prover; the second part is analysis *independent* and it was shown by hand once and for all. For the analysis-dependent part, we define a sufficient soundness property that must be satisfied by each propagation rule in isolation, and we ask an automatic theorem prover to discharge this property for each rule. Separately, we have shown manually that if all propagation rules satisfy the soundness property, then the dataflow information computed by the analysis $\mathcal{A}$ is sound. The formalization of Rhodium, including this manual proof, employs our previous abstract-interpretation-based framework for composing dataflow analyses and transformations [18]. As a result, all Rhodium analyses and transformations can be composed soundly, while allowing them to interact in mutually beneficial ways.

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, σ, with the intent that whenever f appears on an edge, the meaning of f should hold in all concrete execution states σ that can appear on that edge. For example, the meaning of $mustNotPointTo(X, Y)$, shown on line 3 of the Rhodium code, is $\sigma(X) \neq \sigma(\&Y)$, where $\sigma(E)$ represents the result of evaluating expression E in execution state σ. The meaning of *mustNotPointTo* therefore says that the value of X in the execution state σ should not be equal to the address of Y. We denote the meaning of a fact f by $[\![f]\!]$, so that for example $[\![mustNotPointTo]\!](X, Y, \sigma) \triangleq \sigma(X) \neq \sigma(\&Y)$.

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 of execution states right before n, then the meaning of the propagated fact must hold for execution states right after n. To define this formally, we denote by *State* the set of concrete execution states σ, and we use $\sigma \xrightarrow{n} \sigma'$ to say that the execution of n from state σ yields state σ'. We also use $allMeaningsHold(d, \sigma)$ to say that the meanings of all facts in d hold of a program state σ:

$$allMeaningsHold(d, \sigma) \triangleq \forall f(\overrightarrow{t}) \in d \ . \ [\![f]\!](\overrightarrow{t}, \sigma)$$

The soundness of a propagation rule can then be stated as follows:

DEF 1. *A propagation rule* ***if*** ψ ***then*** $f(\overrightarrow{t})@out$ *is said to be sound iff it satisfies the following property:*

$$\begin{array}{l} \forall (n, \sigma, \sigma', d, \theta) \in Node \times State^2 \times D \times Subst \ . \\ \quad \left[\begin{array}{l} [\![\psi]\!](n, d, \theta) \wedge \sigma \xrightarrow{n} \sigma' \wedge \\ allMeaningsHold(d, \sigma) \end{array} \right] \Rightarrow [\![f]\!](\theta(\overrightarrow{t}), \sigma') \end{array} \quad \text{(prop-ok)}$$

For each propagation rule, we use an automatic theorem prover to discharge (prop-ok). The *allMeaningsHold* assumption provides a one-way link between $[\![\psi]\!](n, d, \theta)$ and meanings of facts: it allows the theorem prover to derive $[\![f]\!](\overrightarrow{t}, \sigma)$ from $f(\overrightarrow{t})@in$, but not the other way around. For example, consider the rule in lines 6-7 of figure 1. We effectively ask the theorem prover to show that if a statement satisfying $mustNotDef(X)$ is executed from a state σ in which $\sigma(X) \neq \sigma(\&Y)$, then $\sigma'(X) \neq \sigma'(\&Y)$ in the resulting state σ'. The truth of this formula follows easily from the user-provided definition of *mustNotDef* and the system-provided concrete semantics of our 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 our framework on composing dataflow analyses [18]. This definition depends on an *abstraction function* $\alpha : D_c \rightarrow D$, which formalizes the notion of approximation. The concrete semantics of our 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, taken from [18], is then as follows (where F_c is the concrete collecting semantics flow function):

DEF 2. *A flow function F is said to be sound iff it satisfies the following property:*

$$\begin{array}{l} \forall \ (n, c, d) \in Node \times D_c \times D \ . \\ \quad \alpha(c) \sqsubseteq d \Rightarrow \alpha(F_c(n, c)) \sqsubseteq F(n, d) \end{array}$$

The following lemma, which is proved in the accompanying technical report [20], formalizes 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 lemma from our framework on composing dataflow analyses to show that the analysis $\mathcal{A}$ is sound (where we denote by E_P the set of edges in IL procedure P):

DEF 3. *An analysis $\mathcal{A}$ is said be sound iff for any IL program P, the concrete solution $S_c : E_P \to D_c$ and the abstract solution $S_{\mathcal{A}} : E_P \to D$ satisfy the following property: $\forall e \in E_P \ . \ \alpha(S_c(e)) \sqsubseteq S_{\mathcal{A}}(e)$.*

LEMMA 2. *If the flow function F is sound, then the analysis $\mathcal{A}$ induced by the standard dataflow equations of F is sound.*

A proof of lemma 2 can be found in the accompanying technical report [20]. The following theorem is immediate from lemmas 1 and 2:

THEOREM 1. *If all propagation rules are sound, then the analysis $\mathcal{A}$ induced by the propagation rules is sound.*

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

2.3 Comparison with Cobalt

To better explain the additional expressive power of Rhodium, we show the Cobalt version of the pointer analysis from figure 1:

```
decl $X{:}\,Var,\ Y{:}\,Var,\ Z{:}\,Var$
    $stmt(X := \&Z) \land Y \neq Z$
followed by
    $mustNotDef(X)$
defines
    $mustNotPointTo(X, Y)$
with witness
    $\sigma(X) \neq \sigma(\&Y)$
```

The Cobalt version says that an edge e should be annotated with the *mustNotPointTo*(X, Y) fact if on all CFG paths reaching e, there exists a statement $X := \&Z$ where $Y \neq Z$, which is followed by zero or more statements that do not modify X until the edge e is reached. The region between the statement $X := \&Z$ and the edge e is called the *witnessing region*, and the key property of this region is that the *witness*, in this case $\sigma(X) \neq \sigma(\&Y)$, holds of all program states σ in the region.

As shown above, the condition for triggering a Cobalt transformation is expressed as a global temporal-logic predicate over the entire control flow graph (CFG). This stylized global condition codifies a scenario common to many dataflow analyses: an *enabling* statement establishes a dataflow fact, and then a sequence of zero or more *innocuous* statements preserve it. The Cobalt proof strategy was tailored toward such analyses: we asked the theorem prover to show that the witness was established by the enabling statement and preserved by any innocuous statements. In the pointer-analysis example, the theorem prover would be asked to show that $\sigma(X) \neq \sigma(\&Y)$ holds after a statement $X := \&Z$, where $Y \neq Z$, and that $\sigma(X) \neq \sigma(\&Y)$ is preserved by statements that don't modify X.

While Cobalt can express this analysis and prove it sound automatically, Cobalt's global condition for expressing optimizations has drawbacks. First, Cobalt's proof strategy only allows each dataflow fact to have one associated global condition. This requirement makes it difficult to extend an existing Cobalt analysis. In contrast, a Rhodium analysis can be easily and modularly extended simply by writing new propagation rules.

Second, Cobalt's global condition requires the same dataflow fact to hold throughout the entire witnessing region. In contrast, the Rhodium abstract interpretation strategy allows fine-grained control over how facts are propagated. Programmers can write propagation rules that string different dataflow facts together in flexible ways. This allows Rhodium to express many kinds of global conditions not supported by Cobalt.

Third, Cobalt's metatheory did not allow an analysis to refer to itself, either directly or indirectly. One consequence of this restriction is that the *mustNotDef* fact used in our pointer analysis had to be overly conservative because it could not make use of the pointer information currently being computed. In contrast, the antecedents of Rhodium rules can refer to arbitrary facts, even those that are being propagated in the consequent. The fixed-point semantics of Rhodium and the accompanying abstract interpretation theory ensure that such recursion is well-defined.

To illustrate some of the additional flexibility of Rhodium, we extend our simple pointer analysis from figure 1 with additional rules, slowly building up toward a flow-sensitive version of Andersen's points-to analysis [3]. This analysis was not expressible in Cobalt. We start with a rule for propagating pointer information through simple assignments:

```
decl $X{:}\,Var,\ Y{:}\,Var,\ A{:}\,Var$
if $stmt(X := A) \land mustNotPointTo(A, Y)@in$
then $mustNotPointTo(X, Y)@out$
```

The outgoing information, *mustNotPointTo*(X, Y), is a different instantiation of the *mustNotPointTo* fact than the incoming information, *mustNotPointTo*(A, Y). This way of stringing together *mustNotPointTo*(X, Y) and *mustNotPointTo*(A, Y) was impossible to achieve in Cobalt.

Next we extend our Rhodium analysis with a rule for propagating pointer information through pointer stores:

```
decl $X{:}\,Var,\ Y{:}\,Var,\ A{:}\,Var,\ B{:}\,Var$
if $stmt(*A := B) \land mustPointTo(A, X)@in\ \land$
    $mustNotPointTo(B, Y)@in$
then $mustNotPointTo(X, Y)@out$
```

The *mustPointTo*(A, X) fact, computed by rules not shown here, says that A definitely points to X, and its meaning is $\sigma(A) = \sigma(\&X)$.

The above rule for pointer stores performs a strong update in which we know exactly what A points to. We can also write a weak-update rule for pointer stores:

```
decl $X$: Var, $Y$: Var, $A$: Var, $B$: Var
if stmt$(*A := B) \wedge mustNotPointTo(X, Y)@in \wedge$
    $mustNotPointTo(B, Y)@in$
then $mustNotPointTo(X, Y)@out$
```

Finally, we add a rule for propagating pointer information through pointer loads:

```
decl $X$: Var, $Y$: Var, $A$: Var
if stmt$(X := *A) \wedge$
    $mustNotPointToHeap(A)@in \wedge$
   $\forall B$: Var . $mayPointTo(A, B)@in \Rightarrow$
                $mustNotPointTo(B, Y)@in$
then $mustNotPointTo(X, Y)@out$
```

The $mustNotPointToHeap(A)$ fact, whose rules are not shown here, says that A does not point to the heap (or equivalently, that A points to some variable), and its meaning is $\exists Z : Var \,.\, \sigma(A) = \sigma(\&Z)$. The *mayPointTo* fact is a virtual dataflow fact as defined earlier: $mayPointTo(X, Y) \triangleq \neg mustNotPointTo(X, Y)$. 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.

Starting with a simple pointer analysis 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. Rhodium's propagation rules are the key enablers of this expressiveness leap over Cobalt. Propagation rules allow us to define *mustNotPointTo* recursively, and they allow us to string together instances of the *mustNotPointTo* fact, and other facts, in flexible ways. Rhodium's new proof strategy allows us to automatically prove this analysis sound, despite the extra expressiveness over Cobalt. In section 3 we will show how to extend our Rhodium pointer analysis even further by adding heap summaries, and in sections 5 and 6 we will show how to make it flow-insensitive and/or interprocedural, all while retaining automated soundness reasoning.

2.4 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.

To illustrate transformations, figure 2 shows an arithmetic simplification optimization. The 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* dataflow fact. Some of the rules for this analysis are shown in figure 2. The optimization per se is performed by a single transformation rule on lines 27-28, 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.

We want to automatically show that a Rhodium optimization is sound, according to the following definition:

Def 4. *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 some subset of the transformations suggested by O, has the same semantics as P.*

```
8.  decl $E_1$:Expr, $E_2$:Expr, $E_3$:Expr
9.  decl $X$: Var, $Y$: Var, $I$: Var
10. decl $C$:Int, $C_1$:Int, $C_2$:Int, $C_3$:Int

11. define edge fact $equalsTimes(E_1$:Expr, $E_2$:Expr,
12.                                $E_3$:Expr
13. with meaning $\sigma(E_1) = \sigma(E_2) * \sigma(E_3)$

14. if $equalsTimes(E_1, E_2, E_3)@in \wedge$
15.     $unchanged(E_1) \wedge unchanged(E_2) \wedge$
16.     $unchanged(E_3)$
17. then $equalsTimes(E_1, E_2, E_3)@out$

18. if stmt$(X := I * C) \wedge X \neq I$
19. then $equalsTimes(X, I, C)@out$

20. if stmt$(I := I + C_1) \wedge X \neq I \wedge$
21.     $equalsTimes(X, I, C_2)@in$
22. then $equalsTimes(X, I - C_1, C_2)@out$

23. if stmt$(X := X + C_1) \wedge X \neq I \wedge$
24.     $equalsTimes(X, I - C_2, C_3)@in \wedge$
25.     $C_1 = applyBinaryOp(*, C_2, C_3)$
26. then $equalsTimes(X, I, C_3)@out$

27. if stmt$(Y := I * C) \wedge equalsTimes(X, I, C)@in$
28. then transform $Y := X$
```

Figure 2: Arithmetic simplification optimization in Rhodium. Due to space limitations, only a few representative rules are shown here.

As with propagation rules, our automatic proof strategy requires an automatic theorem prover to discharge a local soundness property for each transformation rule. This property is given in the following definition of soundness for a transformation rule.

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

$$\forall(n, \sigma, \sigma', d, \theta) \in Node \times State^2 \times D \times Subst \;.$$
$$\begin{bmatrix} [\![\psi]\!](n, d, \theta) \wedge \sigma \xrightarrow{n} \sigma' \wedge \\ allMeaningsHold(d, \sigma) \end{bmatrix} \Rightarrow \sigma \xrightarrow{n'} \sigma'$$

The following theorem, which is proven in the accompanying technical report [20], summarizes the part of the proof of soundness of an optimization O that is performed by hand:

Theorem 2. *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 any subset of the suggested transformations can be performed without changing the semantics of any IL procedure.

2.5 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

```
i := 0;
while (...) {
  ...
  i := i + 1;
  ...
  if (...) {
    i := i + 1;
  }
  ...
  y := i * 20;
}
```

(a)

```
i := 0;
x := i * 20;      ⇐ inserted
while (...) {
  ...
  i := i + 1;
  x := x + 20;    ⇐ inserted
  ...
  if (...) {
    i := i + 1;
    x := x + 20;  ⇐ inserted
  }
  ...
  y := x;         ⇐ transformed
}
```

(b)

Figure 3: Code snippet before and after loop-induction-variable strength reduction.

true, and as a result, they can always be safely added to the CFG. We can therefore give programmers a lot of freedom in computing these facts. In particular, we allow 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 are, what the loop-nest 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.

To illustrate the use of profitability facts, we 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 3(a). The result of performing loop-induction-variable strength reduction is shown in figure 3(b). This optimization was not expressible in Cobalt.

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 in order to determine that `x = i * 20` holds just before the statement `y := i * 20`, thereby justifying the strength-reduction transformation. A dead-assignment elimination pass can also be run afterwards in order to clean up the dead assignments to `i`.

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. 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:

- $\mathit{indVar}(I, X, C)$ is inserted on the edges of a loop (including 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.

- $\mathit{afterIncr}(I)$ is inserted on the immediate edge following a statement $I := I + 1$.

- $\mathit{afterLoopInit}(I)$ is inserted on the immediate edge following a statement $I := E$ that is at the head of a loop.

In the example of figure 3, $\mathit{indVar}(\mathtt{i}, \mathtt{x}, 20)$ would be inserted throughout the loop, $\mathit{afterIncr}(\mathtt{i})$ would be inserted after the increments of `i` and $\mathit{afterLoopInit}(\mathtt{i})$ would be inserted after the assignment `i := 0`. The following two transformation rules then indicate which assignments should be inserted:

```
decl X: Var, I: Var, C: Const
if stmt(skip) ∧ dead(X)@out ∧
    afterIncr(I)@in ∧ indVar(I, X, C)@in
then transform X := X + C
if stmt(skip) ∧ dead(X)@out ∧
    afterLoopInit(I)@in ∧ indVar(I, X, C)@in
then transform X := I * C
```

Following our previous work on Cobalt, we express insertion 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 $\mathit{dead}(X)$ 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.[2]

Rhodium's way of incorporating profitability information is superior to Cobalt's approach. Cobalt allowed profitability decisions to be made in a *choose* function that did not affect soundness: after the set of all legal transformations was generated, the *choose* function would select a subset of these transformations to actually perform. The generate-and-test approach of the *choose* function is not always well-suited in practice because there may be infinitely many legal transformations to generate. The above example is such a case: there are infinitely many expressions E for which we can insert an assignments $X := E$ when X is dead. Rhodium solves this problem by allowing programmers to write arbitrarily complex compiler passes for inserting profitability facts that can then be used to prune out the transformations at the point where they are generated.

For the second pass that runs after the dead assignments have been inserted, we can use the arithmetic-invariant analysis from figure 2. The rules in figure 2 are sufficient to trigger the strength-reduction transformation in figure 3(b). The statement `x := i * 20` establishes the dataflow fact $\mathit{equalsTimes}(\mathtt{x}, \mathtt{i}, 20)$. Every sequence of `i := i + 1` followed by `x := x + 20` propagates first $\mathit{equalsTimes}(\mathtt{x}, \mathtt{i\text{-}1}, 20)$ and then $\mathit{equalsTimes}(\mathtt{x}, \mathtt{i}, 20)$. As a result, $\mathit{equalsTimes}(\mathtt{x}, \mathtt{i}, 20)$ is propagated to `y := i * 20`, thereby triggering the transformation to `y := x`.

[2]This example uses the backward dataflow fact $\mathit{dead}(X)$. Section 8 describes the state of backward analyses and transformations in Rhodium.

3. DYNAMIC SEMANTICS EXTENSIONS

The meaning of dataflow facts we have seen 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 talk about complete traces of program states rather than single program states.

As a motivating example, consider extending our pointer analysis from section 2.1 with heap summaries, 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.

We could try to fix this problem by enriching our 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 := `new` T in the trace, the value of X in the successor state). However, in order to extract this information, one has to use quantifiers that range over indices of unbounded-length traces. Unfortunately, we 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.

In order 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. 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 our 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.

We present state extensions in more detail by showing how they can be used to extend our pointer analysis with heap summaries. In order to define the meaning of *mustNotPointTo* over summaries, we define an additional component of the program state called *summary_of*, which maps each heap location to the heap summary that represents it. We 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:

```
decl X: Var, T: Type
if stmt(X := new T)
then (σ@out).summary_of =
        (σ@in).summary_of[σ@out(X) ↦ currNode]
```

The terms $\sigma@in$ and $\sigma@out$ refer respectively to the program states before and after the current statement, while the special term *currNode* refers to the current CFG 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 $\sigma@out$, map to the CFG node that was just executed. In all other cases the *summary_of* component implicitly remains unchanged.

We can easily modify the above declarations to achieve other kinds of summaries. In particular, table 1 shows how to modify the *HeapSummary* definition and change what $\sigma@out(X)$ maps to in the dynamic semantics extension in order to specify different summarization strategies. The rest of our 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 | HeapSummary
```

An abstract memory location AL 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.

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

```
define edge fact mustNotPointTo(AL₁:AbsLoc,
                                AL₂:AbsLoc)
with meaning
    ∀L : Loc .
        belongsTo(L, AL₁, σ) ∧ isLoc(σ(*L)) ⇒
            ¬belongsTo(σ(*L), AL₂, σ)
define belongsTo(L:Loc, AL:AbsLoc, σ:State) ≜
    isVar(AL) ⇒ [L = σ(&AL)] ∧
    isHeapSummary(AL) ⇒ [σ.summary_of[L] = AL]
```

The meaning of *mustNotPointTo* says that none of the locations belonging to AL_1 point to any of the locations belonging to AL_2. The locations belonging to AL_1 are those locations L for which $belongsTo(L, AL_1, \sigma)$ holds. For all these locations L, we look up the memory content of L using $\sigma(*L)$. If the memory content $\sigma(*L)$ is a location (as opposed to a scalar value, which cannot hold pointers), then we want $\sigma(*L)$ to *not* belong to AL_2.

The auxiliary function $belongsTo(L, AL, \sigma)$ returns whether or not a location L belongs to an abstract location AL in state σ. 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 $\sigma.summary\_of$ maps L to AL.

The rules for our pointer analysis must now be modified to take summaries into account. Because of space limitations,

	HeapSummary	$\sigma@out(X)$ 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*	()

Table 1: Various kinds of heap summarization strategies achievable by varying the definition of *HeapSummary* and the dynamic semantics extension.

we only present some representative rules here. The complete set of rules can be found in the accompanying technical report [20].

The following rule, which only works 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 $isLoc(\sigma(X)) \Rightarrow \sigma.summary\_of[\sigma(X)] \neq Summary$. Since the theorem prover knows that `new` T returns a location, it determines that $isLoc(\sigma(X))$ holds, and then the remaining obligation is $\sigma.summary\_of[\sigma(X)] \neq Summary$. To prove this, the theorem prover makes use of the user-defined extension to the dynamic semantics. Indeed, if we let σ be the program state right after executing the allocation, then the dynamic semantics extension tells us that $\sigma.summary\_of[\sigma(X)] = currNode$. In conjunction with $Summary \neq currNode$, this implies $\sigma.summary\_of[\sigma(X)] \neq Summary$, which is what needed to be shown.

The above rule for $stmt(X := new\ T)$ only works for allocation site summaries. Of all the pointer analysis rules, it is the only one that depends on the heap summarization strategy. In order to modify it for other kinds of heap summaries, the antecedent of the rule should compare *Summary* with the third column of table 1, rather than with *currNode*.

Finally, we now show the rule of our pointer analysis that requires the most complicated reasoning from the theorem prover:

```
decl $X{:}Var,\ Y{:}Var,\ AL_2{:}AbsLoc$
if $stmt(X := *Y) \wedge$
    $\forall AL_1 : AbsLoc\ .\ mayPointTo(Y, AL_1)@in \Rightarrow$
                          $mustNotPointTo(AL_1, AL_2)@in$
then $mustNotPointTo(X, AL_2)@out$
```

In the above rule, we again define *mayPointTo* 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 AL_2 after $X := *Y$ if for all abstract locations AL_1 that Y may point to, we have that AL_1 does not point to AL_2.

4. INFINITE ANALYSIS DOMAINS

The domains of dataflow fact parameters in Cobalt were finite for a particular intermediate language program. For example, the *Const* and *Expr* domains did not represent all possible constants and expressions, but rather only those constants and expressions that appeared in the intermediate-language program being analyzed. Rhodium improves on Cobalt by introducing infinite domains. The *Expr* and *Const* domains in Rhodium now refer to the infinite unrestricted versions whereas *ExprInProg* and *ConstInProg* refer to the finite versions restricted to constants and expressions in the source program.[3]

The addition of infinite domains increases the expressiveness of Rhodium. 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 section 2.4. Rhodium can also perform range analysis where the end points of the range are not restricted to constants in the program. Finally, Rhodium can express a better version of constant propagation because it can construct and then propagate constants that are not in the source code.

However, with this extra flexibility comes a challenge: whereas Cobalt analyses were trivially guaranteed to terminate, because all domains were finite, Rhodium analyses may now 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. We deal with each one of these in the next two subsections.

4.1 Termination of a single rule

In order to guarantee that execution of each rule terminates, we 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, we restrict the logic of each rule's antecedent to the decidable subset of first-order logic in which quantifiers only range over finite domains.[4]

For the former, we wish to ensure 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:

[3]When we say finite here, we mean finite *once* a given intermediate-language program has been singled out.

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

```
***define edge fact*** $inRange(X : Var, lo : Const, hi : Const)$
***with meaning*** $lo \leq \sigma(X) \wedge \sigma(X) \leq hi$
***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.

In order to guarantee that such a situation does not occur, we make use of a notion from database community called *safety* [33], adapting it to the context of Rhodium. 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.

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 $\perp$. Since it is sound to propagate $\perp$ as the result of any rule invoked with $\perp$ on its input, we treat $\perp$ specially and directly propagate $\perp$ to the output without invoking the rule explicitly.

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

4.2 Termination of the fixed-point computation

As discussed in section 2.1, 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, we recall from section 4.1 that we already imposed the *finite-safe* requirement, which led to all propagated sets being either finite or $\perp$. We can therefore shrink our lattice to only include 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.

Our technique for guaranteeing termination is effective even in the face of dataflow facts with infinite-domain parameters. For example, the *equalsTimes* dataflow fact has all three of its parameters ranging over infinite domains, and yet we are 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.

4.3 Custom merges

The range-analysis propagation rule in section 4.1 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 we use intersection to join the fact set $\{inRange(\mathtt{x}, 1, 1)\}$ with $\{inRange(\mathtt{x}, 2, 2)\}$, we get $\{\}$. We 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, we 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 example introduce edge indices: $in[i]$ refers to the i^{th} CFG input edge. The previously used *in* was just syntactic sugar for $in[0]$. Similarly, *out* can also be indexed to refer to the true and false successor edges of a branch node. 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-ok) that handles an arbitrary number of input and output edges is given in the accompanying technical report [20]. 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, we would ask the theorem prover to show that if the meaning of $inRange(X, C_1, C_2)$ holds of some program state σ, and σ on edge 0 steps to σ' through the merge node, then the meaning of $inRange(X, min(C_1, C_3), max(C_2, C_4))$ holds of σ'. 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, we instead need a lattice in which $S \sqcup T \sqsubseteq merge(S, T)$ 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, we make use of the more general lattice of predicates: $(D, \sqcup, \sqcap, \sqsubseteq, \top, \perp) = (Pred, \vee, \wedge, \Rightarrow, true, false)$. 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, we interpret these sets 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-ok), we are 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 $\perp$, does not have the finite-ascending-chain property. Consider for example the *inRange* fact, 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 analyses using custom merges, and indeed the kind of range analysis discussed here does not terminate.[5]

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 [8]. 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. 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 is run on n, d_{in}, and d_{out} to produce the widened output set d_{wide}. Finally, we compute $merge(d_{out}, d_{wide})$ (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 fact d_{out} is sound, and since $merge(d_{out}, d_{wide})$ is more conservative than d_{out}, $merge(d_{out}, d_{wide})$ must also be sound, which 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.

5. FLOW-INSENSITIVE ANALYSES

An additional benefit that falls out from Rhodium's new flow-function model is that Rhodium can easily support provably sound flow-insensitive analyses. In particular, we can interpret propagation rules in a flow-insensitive manner. Instead of keeping a separate set of dataflow facts at each edge, we keep 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. We have shown once by hand that if all the propagation rules are sound, then the result of running the analysis in flow-insensitive mode is also sound. A proof can be found in the accompanying technical report [20].

6. INTERPROCEDURAL ANALYSES

Yet another benefit of using flow functions is that we can adapt a previous flow-function-based framework [7] from the Vortex compiler [11] in order to automatically build provably sound interprocedural analyses in Rhodium. The previous Vortex framework has been used to write realistic interprocedural analyses, such as various kinds of class analysis [13], constant propagation, side-effect analysis, escape analysis, and various synchronization-related analyses [2]. The contribution of the new Rhodium framework is a rigorous formal description combined with a proof of soundness. These are stand-alone contributions whose applications are broader than just the Rhodium system.

Our approach revolves around a framework for creating a provably sound interprocedural analysis from a sound intraprocedural version. The 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 $c \in Context$ in which the caller is being analyzed, and the dataflow information d at the call site, $selectCalleeContext(n, c, d)$ returns the context for analyzing the callee at this call site.

We have instantiated our framework with two commonly used context-sensitivity strategies: the transfer function strategy (also known as Sharir and Pnueli's functional approach [27]), and Shivers's k-CFA algorithm [28] (also known as the k-deep call-strings strategy of Sharir and Pnueli [27]). Table 2 shows the definition of *Context* and *selectCalleeContext* for these two strategies. The context-insensitive strategy can be achieved using 0-CFA.

Our key insight is that these instantiations of the framework can be proven sound by hand once and for all, independent of any user-defined analysis. As a result, any interprocedural analysis generated by one of these instantiations is guaranteed to be sound provided the intraprocedural version is. To build a provably sound interprocedural analysis, the programmer 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. Our framework then automatically generates an interprocedural version of the analysis that is guaranteed to be sound.

The Rhodium framework operates by creating an interprocedural flow function F_i from an intraprocedural version F. Due to space limitations, we only give an informal description of F_i here – a formal description of the framework, accompanied by proofs of soundness, can be found in the accompanying technical report [20].

Instead of propagating facts $d \in D$, the interprocedural analysis propagates partial maps $cd \in Context \rightharpoonup D$ which map a calling context c to the dataflow information d that holds in that context. 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. The accompanying technical report [20] describes how dataflow facts are translated from callers to callees and vice versa.

Analogously to widening operators as discussed in sec-

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

Strategy	*Context*	*selectCalleeContext*
Transfer function	D	$selectCalleeContext(n, c, d) = d$
k-CFA	$list[string]$	$selectCalleeContext(n, c, d) = last(concat(c, [fnOf(n)]), k)$ where: $concat(l_1, l_2)$ 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)

Table 2: Definition of *Context* and *selectCalleeContext* for two common context-sensitivity strategies.

tion 4.3, we could 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 $\top$, bounding the number of times the function is analyzed.

7. 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 [18], we have implemented a forward intraprocedural execution engine for the core of the Rhodium language. Rhodium optimizations in Whirlwind peacefully co-exist with Cobalt optimizations and with hand-written 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 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 $F_{exec} : Node \times D \to D$ operates as follows (where R is the set of forward propagation rules that the engine is executing):

$$\begin{array}{l} F_{exec}(n, d) = \cup_{r \in R}\ apply\_rule(r, n, d) \\ apply\_rule(\textbf{\textit{if}}\ \psi\ \textbf{\textit{then}}\ f(\overrightarrow{t})@out, n, d) = \\ \quad \textbf{let}\ \Theta = sat(\psi, n, d, []) \ \textbf{in}\ \cup_{\theta \in \Theta} \{f(\theta(\overrightarrow{t}))\} \end{array}$$

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, $f(\theta(\overrightarrow{t}))$, to the result set.

The $sat : Pred \times Node \times D \times Subst \to 2^{Subst}$ 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 set of facts d, and a substitution θ, $sat(\psi, n, d, \theta)$ returns the set of all substitutions θ' that have the following properties: (1) θ' makes ψ hold at node n when d flows into n, or more formally, $[\![\psi]\!](n, d, \theta')$ 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 $sat(\psi, n, d, [])$ 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*:

$$\begin{array}{lcl} sat(true, n, d, \theta) & = & \{\theta\} \\ sat(false, n, d, \theta) & = & \emptyset \\ sat(\psi_1 \vee \psi_2, n, d, \theta) & = & sat(\psi_1, n, d, \theta) \cup sat(\psi_2, n, d, \theta) \\ sat(\psi_1 \wedge \psi_2, n, d, \theta) & = & \textbf{let}\ \Theta = sat(\psi_1, n, d, \theta) \\ & & \quad \textbf{in}\ \cup_{\theta' \in \Theta}\ sat(\psi_2, n, d, \theta') \\ sat(t_1 = t_2, n, d, \theta) & = & unify(n, t_1, t_2, \theta) \\ sat(f(\overrightarrow{t})@in, n, d, \theta) & = & \cup_{f(\overrightarrow{s}) \in d}\ unify\_terms(n, \overrightarrow{t}, \overrightarrow{s}, \theta) \\ sat(\exists x.\psi, n, d, \theta) & = & sat(\psi, n, d, \theta \setminus x)[x \mapsto \theta(x)] \end{array}$$

In the above definition, we use $\theta \setminus x$ to denote θ with any mapping of x removed. We 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 $\theta'[x \mapsto \theta(x)]$.

The *sat* function above 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 the empty set is returned. If the unification succeeds with substitution θ', then θ' is augmented with all the mappings from θ to produce θ'', and the singleton set $\{\theta''\}$ is returned. The *unify_terms* function works like *unify*, except that it unifies a sequence of terms $\overrightarrow{t}$ with another sequence $\overrightarrow{s}$. The unification procedure also tries to evaluate terms such as $applyBinaryOp(*, C_2, C_3)$ from figure 2. 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.

8. CURRENT AND FUTURE WORK

We have so far focused our attention primarily on forward analyses and transformations in Rhodium. We have implemented a fully automated checker and execution engine for Rhodium forward analyses and transformations, and we have finished the hand proofs for the forward case.

We are now extending our work to backward optimizations. We already have a proof strategy for backward Rhodium analyses and transformations, but have not yet implemented the checker nor completed the hand proofs.

We have written in Rhodium the two backward optimizations we had in Cobalt (dead assignment elimination and code hoisting), and simulated our proof strategy by hand on these optimizations. The proof obligations for these two optimizations in Rhodium end up being exactly the same as the proof obligations for their Cobalt counterparts. We are currently working on generating these proof obligations mechanically, and we are also in the process of finishing the hand proofs for the backward case.

In future work, we would like to extend our execution engine to handle the full language design, including backward analyses and transformations, interprocedural and flow-insensitive analyses, profitability heuristics and user-defined widenings.

We also plan to explore more efficient implementation strategies for our execution engine, such as generating specialized code to run each optimization [30]. For example, consider a rule whose antecedent is a conjunction where one of the conjuncts is $\mathit{stmt}(X := \&Z)$. We statically know that this rule will only fire on statements of the form $X := \&Z$, but because our current engine does not make use of this information, the rule is repeatedly considered on statements of the "wrong" form. By partially evaluating the rules with respect to each statement kind, we can produce a specialized set of rules that will be smaller than the whole set (because some rules will not apply) and in which each rule will be simpler (because the antecedent can be simplified based on the statement kind). The generated flow function would dispatch on the form of the statement being analyzed, and would directly jump to specialized code that runs the simplified rules.

Furthermore, we would also like to investigate more efficient representations of the dataflow information. For example, storing the does-not-point-to relation using explicit pairs can incur a significant memory overhead. We would like to investigate ways of automatically converting to more space-efficient representations, for instance the inverted may-point-to relation, or a bit-vector representation of the relation. Also, motivated by recent advances in the use of BDDs to represent pointer information [5, 34], we would like to explore ways of inferring when it would be beneficial to use BDDs for encoding our sets of facts.

Finally, we want to continue on our path of pushing more and more of the burden of compiler-writing onto the computer. By automating more and more of the tedious, difficult and error-prone parts of compiler-writing, we can allow the human to concentrate on the creative and interesting parts. One such direction is to automatically infer propagation rules given only the facts and their meanings. Another direction would be to generate the facts, meanings and propagation rules for supporting a given CFG rewrite rule.

9. RELATED WORK

The idea of analyzing optimizations written in a domain-specific language was introduced by Whitfield and Soffa with the Gospel language [35]. The differences between our work and the Gospel work stem from the difference in focus: we explore soundness whereas Whitfield and Soffa explore optimization dependencies.

Many other frameworks and languages have been proposed for specifying dataflow analyses and transformations, including Sharlit [32], System-Z [36], languages based on regular path queries [29], and temporal logic [30, 17]. None of these approaches, however, addresses automated soundness checking of the specified transformations.

A significant amount of work has been done on manually proving dataflow analyses and transformations correct, including abstract interpretation [8, 9, 10], the work on the VLISP compiler [14], Kleene algebra with tests [16], manual proofs of correctness for optimizations expressed in temporal logic [30, 17], and manual proofs of correctness based on partial equivalence relations [4]. Analyses and transformations have also been proven correct mechanically, but not automatically: the soundness proof is performed with an interactive theorem prover that requires guidance from the user. For example, Young [37] has proven a code generator correct using the Boyer-Moore theorem prover enhanced with an interactive interface [15]. As another example, Cachera et. al. [6] 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.

Instead of proving that the compiler is always correct, translation validation [25, 23] and credible compilation [26] both attack the problem of checking the correctness of a given compilation run. Therefore, a bug in an optimization only appears when the compiler is run on a program that triggers the bug. Our work allows optimizations to be proven correct before the compiler is even run once. However, to do so we require optimizations to be written in a special-purpose language. Our approach also requires the Rhodium execution engine to be part of the trusted computing base, while translation validation and credible compilation do not require trust in any part of the optimizer.

Proof-carrying code [22], certified compilation [24], typed intermediate languages [31], and typed assembly languages [21] have all been used to prove properties of programs generated by a compiler. However, the kinds of properties that these approaches have typically guaranteed are type safety and memory safety. In our work, we prove the stronger property of semantic equivalence between the original and resulting programs.

10. CONCLUSION

We presented a new language called Rhodium for expressing dataflow analyses and transformations that is significantly more expressive than previous work while retaining automated soundness checking. The key to Rhodium's expressiveness lies in its use of local propagation rules, which can be used by programmers to implement flow functions that are checked automatically for soundness, and from which can be derived flow-insensitive, flow-sensitive, and interprocedural analyses.

Acknowledgments

This research is supported in part by NSF grants CCR-0073379, ACI-0203908 and ITR-0326590, a Microsoft Graduate Fellowship, an IBM Faculty Development Award, and by gifts from Sun Microsystems. We would also like to thank Dan Grossman, Alexandru Sălcianu, and the anonymous re-

viewers for their useful suggestions on how to improve the paper.

11. REFERENCES

[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] Jonathan Aldrich, Craig Chambers, Emin Gun 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, 1999.

[3] 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).

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

[5] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig 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.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, June 2002.

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

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

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

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

[17] 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 OR, January 2002.

[18] 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.

[19] 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*, pages 220–231. ACM Press, 2003.

[20] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. Technical Report UW-CSE-2004-07-04, University of Washington, July 2004.

[21] 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.

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

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

[24] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[25] 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.

[26] Martin Rinard and Darko Marinov. Credible compilation. In *Proceedings of the FLoC Workshop Run-Time Result Verification*, July 1999.

[27] 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.

[28] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta GA, June 1988.

[29] 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.

[30] 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.

[31] 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 '96 Conference on Programming Language Design and Implementation*, Philadelphia PA, May 1996.

[32] Steven W. K. Tjiang and John L. Hennessy. Sharlit – a tool for building optimizers. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 82–93, July 1992.

[33] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Volume I.* Computer Science Press, 1988.

[34] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation.* ACM Press, June 2004.

[35] 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.

[36] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

[37] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.