Automatic Inference of Optimizer Flow Functions from Semantic Meanings

Erika Rice Scherpelz
Google
erikars@google.com

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

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

Abstract

Previous work presented a language called Rhodium for writing program analyses and transformations, in the form of declarative flow functions that propagate instances of user-defined dataflow fact schemas. Each dataflow fact schema specifies a semantic meaning, which allows the Rhodium system to automatically verify the correctness of the user's flow functions. In this work, we have reversed the roles of the flow functions and semantic meanings: rather than *checking* the correctness of the user-written flow functions using the facts' semantic meanings, we automatically *infer* correct flow functions solely from the meanings of the dataflow fact schemas. We have implemented our algorithm for inferring flow functions from fact schemas in the context of the Whirlwind compiler, and have used this implementation to infer flow functions for a variety of fact schemas. The automatically generated flow functions cover most of the situations covered by an earlier suite of handwritten rules.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – *compilers, optimization*

General Terms Languages

1. Introduction

Compilers are an important part of the computing infrastructure relied upon by software developers. Of course, developers require their compilers to be *correct*, i.e., to translate a source program into a target program that has the same externally visible behavior; an incorrect compiler can fatally compromise any guarantees made by the source program. In addition, developers need their compilers to produce *efficient* target programs. As the level of the source program moves farther above the level of the target program, increasing demands are placed on the compiler's optimizer to achieve this efficiency. Unfortunately, it is hard to develop an optimizer that is both correct and that produces efficient target code. When developing an optimization, it is difficult to reason about the possible effects of all the different kinds of statements in the compiler's intermediate representation, and it is hard to develop test suites that adequately exercise all the combinations of statements and probe the "corner cases." Moreover, an optimizer is comprised of a collection of optimizations, which can interact in subtle ways, further exploding the test space. As a result, it often takes years of testing and use for an optimizing compiler to become mature and reliable, and even so it can still contain numerous, rarely-occurring bugs. Aside from having an impact on software reliability in general, the difficulty of writing correct optimizing compilers also hinders the development of new languages and new architectures, and it discourages end programmers from extending compilers with domain-specific checkers or optimizers.

Our broad research agenda is to provide better support for writing correct, efficient optimizers. Previously we presented a language called Rhodium for writing compiler optimizations that could be checked for correctness automatically [17, 16]. Rhodium users (i.e., compiler developers) declare what kinds of information they want to compute using *dataflow fact schemas*, and then they write *propagation rules* (a.k.a. flow functions) for generating and propagating instances of these fact schemas across intermediate-language statements, as well as *transformation rules* for using inflowing facts to optimize statements. By providing specialized support for the task of writing optimizations, Rhodium makes optimizations easier to write and maintain, which helps in reducing errors. Moreover, the restricted domain makes Rhodium amenable to rigorous static checking that would otherwise be infeasible. In particular, the user provides a simple *semantic meaning* for each dataflow fact schema, which the Rhodium system uses to automatically prove that the user-defined propagation and transformation rules are correct.

Although Rhodium makes it easier to write dataflow analyses and optimizations, the user still has to invest a significant amount of effort writing propagation rules for each kind of fact and for each kind of statement. In this paper, we present a technique for automatically *inferring* correct forward propagation rules given only a set of declared dataflow fact schemas and their associated semantic meanings. In the case of constant propagation, for example, the user would only declare the fact schema

$$hasConstValue(X : Var, C : Const)$$
with meaning $\sigma(X) = \sigma(C)$

which specifies that a dataflow fact of the form $hasConstValue(X, C)$, where X is a variable and C is a constant, can be correctly propagated to a control flow graph edge if, for every possible run-time program state σ that can exist when control reaches that edge, the result of evaluating X equals the result of evaluating C. Our system would then combine the user-defined semantic meaning with the system-provided semantics of each intermediate language statement form to automatically infer correct propagation rules for this fact schema. As a result, the user need not be burdened with writing propagation rules, making it even easier to produce correct dataflow analyses. In effect, we have recast the semantic meanings associated with facts from a cost to a

PLDI'07 June 11–13, 2007, San Diego, California, USA.

benefit: in the original Rhodium system, semantic meanings were extra redundant information provided solely to verify existing rules (and provide formal documentation), while our new algorithm constructively uses the meanings to derive rules automatically.

To automatically generate forward propagation rules for a fact schema, our algorithm works backward from its semantic meaning. For an instance of a fact schema to be correctly introduced on the outgoing edge of a CFG node, the fact's semantic meaning must hold for all possible program states that could arise when control reaches that edge. Our algorithm uses weakest preconditions to compute the condition that must hold *before* the node in order for the fact's semantic meaning to hold *after* the node. Finally, our algorithm applies a series of simplifications and strengthenings to re-express this weakest precondition in terms of some combination of the semantic meanings of user-defined fact schemas.

The contributions of this paper are as follows:

- We show how the problem that we are solving in the context of Rhodium is an instance of the *parametric predicate abstraction* (PPA) problem, a generalization of predicate abstraction [9, 6, 2, 14] introduced by Cousot [5]. The PPA problem has been solved previously, but only by hand, and only for a particular set of parametric predicates. We present an algorithm for inferring correct forward propagation rules from a set of fact schema declarations, in the process solving the PPA problem *automatically* and for *any* set of parametric predicates. (Handwritten propagation rules can still be provided to augment the inferred set, ensuring that any limitations of the inference algorithm do not block its use.) Section 3 presents an overview of our approach, and section 4 presents our algorithm in detail.
- We have implemented our algorithm in the context of the Whirlwind compiler, and we have run it to generate rules for a variety of fact schemas. (In Whirlwind, Rhodium-based optimizations interoperate seamlessly with non-Rhodium optimizations, allowing incremental migration of non-Rhodium optimizations to Rhodium form.) Section 5 describes these fact schemas and the generated rules.
- We have developed a method to automatically compare the *theoretical* coverage of two sets of Rhodium rules, independent of any particular program that the rules may run on. We have used this method to compare the coverage of the automatically generated propagation rules to an earlier set of handwritten rules. For the facts we considered, our automatically generated rules covered 77% of the cases of the handwritten rules, and the automatically generated rules included cases that were not covered by the handwritten rules. Section 5.1 presents a more detailed description of these results.
- We have also compared the *practical* coverage of the automatically generated rules versus the handwritten rules, in terms of the optimizing transformations triggered when the rules are run on a set of benchmark programs. Our results show that the inferred rules were able to trigger over 95% of the transformations triggered by the handwritten rules, and some transformations not triggered by the handwritten rules.

By reducing the burden of writing correct dataflow analyses, we hope that our work will allow compiler writers to focus on the more creative parts of their task, such as determining what novel transformations could improve performance, and what dataflow facts are needed to justify these transformations. Furthermore, by reducing the barrier to entry, our system makes it feasible to open up compilers to user extensions without compromising correctness. Oftentimes, it is the end programmer who has the application-domain knowledge and the coding-pattern knowledge necessary to implement useful analyses and optimizations. The Rhodium rule inferencer could allow end programmers with little knowledge of program analysis to implement such analyses and optimizations without the worry of breaking the compiler.

Stmts	s	::=	$x := e \mid *x := x \mid$ `decl` $x \mid x :=$ `new` $\mid$ `decl` $x[x] \mid x :=$ `new`$[x] \mid$ `if` x `goto` l `else` $l \mid$ `skip`
Exprs	e	::=	$c \mid x \mid *x \mid \ op\ x\ \ldots\ x \mid \&x \mid \&(x[x])$
Ops	op	::=	various operators with arity ≥ 1
Vars	x	::=	`x` $\mid$ `y` $\mid$ `z` $\mid \ldots$
Consts	c	::=	constants
Labels	l	::=	statement labels

Figure 1. Grammar of the IL

```
1.  define edge fact schema hasConstValue(X:Var, C:Const)
2.  with meaning σ(X) = σ(C)

3.  decl X:Var, C:Const

4.  if currStmt=[X := C]
5.  then hasConstValue(X, C)@out

6.  if hasConstValue(X, C)@in ∧ currStmt=[Z := Y] ∧ X ≠ Z
7.  then hasConstValue(X, C)@out

8.  if hasConstValue(X, C)@in ∧ currStmt=[*Z := Y]∧
9.     mustNotPointTo(Z, X)@in
10. then hasConstValue(X, C)@out
```

Figure 2. Simple constant propagation analysis in Rhodium

2. Background on Rhodium

This section presents background material on how dataflow analyses are written in Rhodium, and how they are checked for correctness automatically. Rhodium analyses run over a C-like intermediate language (IL) with functions, recursion, unstructured local control flow, pointers to stack- and heap-allocated memory, and stack- and heap-allocated arrays. The IL program is represented as a control flow graph (CFG) with each node representing a simple register-transfer-level statement, shown in Figure 1. (We currently do not attempt to infer propagation rules for function call statements, so they are omitted from the figure.)

Dataflow information is encoded in Rhodium by means of dataflow facts, which are user-defined function symbols applied to a set of terms, e.g., $hasConstValue(\mathtt{x}, \mathtt{5})$ and $exprIsAvailable(\mathtt{x}, \mathtt{a} + \mathtt{b})$. A Rhodium analysis uses *propagation rules*, which are a stylized way of writing flow functions, to specify how dataflow facts are introduced after and propagated across CFG nodes. These user-defined flow functions collectively define a dataflow analysis, whose solution is the greatest fixed point of the equations induced by the flow functions.

The example in Figure 2 shows a partial implementation of constant propagation in Rhodium. We encode the dataflow information for this analysis using the $hasConstValue(X, C)$ *edge fact schema* declared on line 1. A fact schema is a parameterized dataflow fact (a pattern, in essence) that can be instantiated to create actual dataflow facts. With the declaration on line 1, each edge in the CFG will be annotated with a set containing facts of the form $hasConstValue(X, C)$, where X ranges over variables in the IL program being analyzed, and C ranges over constants. The convention used throughout this paper is that Rhodium meta-variables C and K range over constants while W, X, Y, and Z range over IL variables.

Propagation rules in Rhodium indicate how edge facts are introduced after and propagated across CFG nodes. For example, the rule on lines 4-5 of Figure 2 specifies that a variable is known to have a particular constant value immediately after every statement that assigns the constant to the variable. The rule on lines 6-7 of Figure 2 defines a condition for preserving a $hasConstValue$ fact across an assignment statement: if the fact $hasConstValue(X, C)$ appears on the incoming CFG edge of an assignment and X is not the variable assigned to, then the dataflow fact $hasConstValue(X, C)$ should appear on the outgoing edge of the assignment. Rules can refer to more than one kind of fact as well, allowing the results of multiple analyses to cooperate. For example, the rule on lines 8-10 of Figure 2 leverages points-to analysis (defined elsewhere) to preserve constant propagation information across pointer store statements.

Each propagation rule is interpreted within the context of a CFG node, referred to as $currStmt$. References to edge facts are followed by @ signs, with the name after the @ sign indicating the edge on which the fact appears. The in edge name refers to the incoming edge of the current node, and the out edge name refers to the outgoing edge. For nodes with more than one incoming or outgoing edge, such as conditional branches, edge names can be subscripted. For example, $hasConstValue(X, C)@out[0]$ and $hasConstValue(X, C)@out[1]$ can be used to refer to constant propagation facts along a conditional branch's true and false outgoing edges, respectively.

The part of a rule before the **then** keyword is called the *antecedent* and the part after is called the *consequent*. Intuitively, the antecedent is a first order logic formula over dataflow facts (details of the antecedent syntax can be found elsewhere [15]). The semantics of a propagation rule on a CFG node 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 generated on the named edge. For the rule described above, the $hasConstValue(X, C)$ fact will be generated on the outgoing edge of a node for each substitution of X and C with variables and constants that makes the antecedent valid.

To check Rhodium dataflow analyses for correctness, programmers must specify a semantic meaning for each fact schema, in the form of a first order logic predicate over a program state σ. For example, the meaning of $hasConstValue(X, C)$, shown on line 2, is that the value of X in σ, denoted by $\sigma(X)$, is equal to the value of the constant C, denoted by $\sigma(C)$. This meaning states that if $hasConstValue(X, C)$ appears on an edge in the CFG, then for any σ that may occur at run-time when control reaches that edge, $\sigma(X) = \sigma(C)$ holds.

Rhodium dataflow analyses are checked for correctness automatically by discharging a local correctness obligation for each propagation rule. For each rule an automated theorem prover is asked to show that if the meaning of the antecedent holds before a node for an otherwise-arbitrary program state σ, then the meaning of the consequent holds after executing the node in σ. We have shown by hand that if all the propagation rules in a Rhodium program pass this condition, then the induced dataflow analysis is correct. The full details of the Rhodium language, including syntax for defining dataflow facts and meanings, can be found elsewhere [15].

3. Overview

The Rhodium system makes it easier to write dataflow analyses by providing a convenient way to express flow functions, and by automatically checking their correctness. However, there is still a significant amount of effort that the Rhodium user has to invest to write the flow functions. There are a many statement forms to be considered, and many of the resulting propagation rules can be quite similar. For example, the rule on lines 6-7 of Figure 2 is a typical *preservation* rule, which states the conditions under which $hasConstValue(X, C)$ is preserved through a certain statement form; a similar rule should be written for many other assignment statement forms. Even though many of these rules are simple and straightforward, subtle corner cases can exist, particularly in the face of aliasing and mutation. Moreover, information computed by one analysis can be used to help compute information for another analysis, as illustrated by the rule on lines 8-10; it is difficult (and non-modular) for Rhodium users to think through all the possible beneficial interactions among a set of dataflow analyses. Finally, the programmer has to invest time and effort debugging any rules that fail the correctness checks.

To alleviate these burdens, we wish to automatically infer the most complete set of correct forward propagation rules we can, given only a set of dataflow fact schema declarations. For the example in Figure 2, our goal would be to infer propagation rules such as the ones on lines 4-10 (and many more) from the dataflow fact schema definition on lines 1-2 (and the definitions of other fact schemas such as $mustNotPointTo$).

This problem of inferring Rhodium propagation rules is related to predicate abstraction [9, 6, 2, 14] and its generalization parametric predicate abstraction [5]. The next two subsections describe predicate abstraction and previous algorithms for performing it, and explain how that work is insufficient for solving our problem. The last subsection presents an overview of our approach.

3.1 Predicate Abstraction

The goal of predicate abstraction is to propagate a given set of predicates through a program. Formally, predicate abstraction is defined in terms of a finite set $\mathcal{B} = \{B_1 \ldots B_k\}$ of predicate names. Each predicate name B_i has an interpretation $[\![B_i]\!] = \varphi_i$, which is the predicate for which B_i stands. A cube over $\mathcal{B}$ is a conjunction of possibly negated symbols from $\mathcal{B}$, which can be represented as a set of possibly-negated names drawn from $\mathcal{B}$. The domain of predicate abstraction is the set of all cubes, and one cube is computed at each program point. Given these definitions, the problem of predicate abstraction (PA) can be stated as follows:

PA PROBLEM STATEMENT. Given a set of predicates $\mathcal{B}$ with associated interpretations, a program statement s, and a cube over $\mathcal{B}$ flowing into the statement, determine the cube over $\mathcal{B}$ that flows out of the statement.

For example, consider the set of predicate names $\mathcal{B} = \{B_1, B_2\}$, where $[\![B_1]\!] \triangleq (\mathtt{a} = \mathtt{b})$ and $[\![B_2]\!] \triangleq (\mathtt{a} = \mathtt{b} + 1)$. Given the cube $B_1 \wedge \neg B_2$ and the statement $\mathtt{a} := \mathtt{b} + 1$, then predicate abstraction would compute that the cube $\neg B_1 \wedge B_2$ should be propagated after the statement.

Our problem of inferring Rhodium propagation rules from fact schemas is different from predicate abstraction in a simple yet fundamental way. Although our algorithm starts with a set of named predicates as in predicate abstraction (and these named predicates are called fact schemas in the context of Rhodium), our named predicates are *parametric*. For example, our algorithm doesn't start with concrete predicates, such as $\mathtt{a} = 5$ or $\mathtt{a} = \mathtt{b}$, but rather with *parametric* predicates, such as $X = C$ or $X = Y$ (we temporally omit σ from our meanings here to be consistent with the σ-less notation used in predicate abstraction). These parametric predicates represent an infinite set of instantiated concrete predicates. In addition to predicates being parametric, the statements that our algorithm considers are also parametric. For example, our algorithm doesn't infer flow functions for concrete statements such as $\mathtt{a} := \mathtt{a} + \mathtt{b}$, but rather for parametric statements (or statement *forms*) such as $X := Y\ OP\ Z$. This reflects the fact that our goal is to infer flow functions prior to running the analysis on any particular program.

In this light, the problem that we are solving can be seen as a generalization of predicate abstraction that applies to *parametric predicates*. Such a generalization was proposed by Cousot [5], and he called the generalized problem *parametric* predicate abstraction (PPA). Formally, PPA is defined in terms of a finite set $\mathcal{B} = \{B_1 \ldots B_k\}$ of *parametric* predicate names, where we denote by $args(B_i)$ the sequence of parameter names. Each predicate name B_i has a interpretation $[\![B_i]\!] = \varphi_i$, where any free variables in φ_i must be in $args(B_i)$. Applied to the Rhodium context, a predicate name B_i would correspond to a Rhodium fact schema name, $args(B_i)$ to the parameters of the fact schema, and its interpretation to the meaning of the fact schema. The PPA problem can then be stated as follows:

PPA PROBLEM STATEMENT. Given a set of *parametric* predicates $\mathcal{B}$ with associated interpretations, and given a *parametric* statement (corresponding to a statement form in the intermediate language), determine a representation of the flow function for that statement form.

3.2 Algorithms for Predicate Abstraction

There are a variety of approaches to performing traditional predicate abstraction automatically [9, 6, 2, 11, 14], but they are all based on the observation that, because there are a finite number of predicates, one can test each predicate separately to see if it (or its negation) holds after the statement. This test is usually done by asking a theorem prover to show an implication. For example, to test if a predicate B_i holds after a statement s given that a cube c holds before the statement, one simply asks an automatic theorem prover to show that the cube c implies $wp(s, B_i)$, the weakest precondition of B_i with respect to s. The cube that is propagated includes all the possibly-negated predicates for which the theorem prover can show this implication. The computation of which cube to propagate can be performed while predicate abstraction is running on a given program, or it can be performed ahead of time, once the program is known, by enumerating all possible incoming cubes to all the statements in the program (which again is possible because the set of predicates is finite).

The approach to predicate abstraction described above is essentially an enumerate-and-test approach. In our context, such an approach would amount to enumerating possible antecedents and checking whether or not each antecedent implies the consequent we want to propagate. For example, consider two fact schemas $varEqualsPlus(X, Y, Z)$ with meaning $\sigma(X) = \sigma(Y) + \sigma(Z)$ and $varNotEqualsPlus(X, Y, Z)$ with meaning $\sigma(X) \neq \sigma(Y) + \sigma(Z)$, and consider the problem of propagating the fact $varEqualsPlus(X, Y, Z)$ on the out edge of the parametric statement $A_1 := A_2 + A_3$. The most natural enumerate-and-test approach would be to enumerate possible predicates ψ involving $varEqualsPlus$ and $varNotEqualsPlus$, and then use the Rhodium correctness checker to see whether or not the rule **if** $currStmt{=}[A_1 := A_2 + A_3] \wedge \psi$ **then** $varEqualsPlus(X, Y, Z)@out$ is correct.

Unfortunately, because the antecedent can have more free variables than appear in the consequent, the number of possible ψ predicates is infinite. Even if one restricts the predicates to only mention meta-variables "bound" by the statement form and consequent fact to propagate, the number of possible predicates grows very fast. Consider the example above, where we are considering propagation rules for the statement $A_1 := A_2 + A_3$ and outgoing fact $varEqualsPlus(X, Y, Z)@out$, whose antecedent can mention instances of the fact schemas $varEqualsPlus$ and $varNotEqualsPlus$ on the in edge. There are a total of 6 meta-variables (X, Y, Z, A_1, A_2, and A_3) "bound" in the statement and the outgoing fact. Since the fact schema $varEqualsPlus(X, Y, Z)$ has three parameters, it can be instantiated in $6^3 = 216$ ways, and similarly for $varNotEqualsPlus$, so there are a total of 432 instantiations of the fact schemas. If we limit ourselves to conjunctions of size three, we would have 432 **choose** 3 = 13,343,760 possible conjunctions to check. With the optimistic assumption that correctness checking of a candidate propagation rule takes about 1 second, this amounts to about 5 months of continuous checking. Furthermore, this simple computation is only an under-approximation, because it includes just two fact schemas; it does not include all the possible equality and inequality constraints between meta-variables (such as the one present on line 6 of Figure 2), and it does not include the possibility of having meta-variables in the antecedent of a rule that are not present in the consequent.

Although there are techniques to reduce the search space for the traditional problem of predicate abstraction [2], the broader problem with the enumerate-and-test approach is simply that it is not goal-directed: it blindly enumerates predicates, without any consideration for the end goal, and then it does a forward check to see if the end goal is met.

Solutions to the parametric predicate abstraction problem would likely be more appropriate to our task of inferring Rhodium propagation rules, but previous solutions to the PPA problem have been constructed by hand, and only for particular sets of parametric predicates. A key contribution of our work is to develop an algorithm that solves the PPA problem *automatically* for *any* given set of parametric predicates.

3.3 Our Approach

The solution we propose in this paper takes a goal-directed approach, which is much better suited to the PPA problem than an enumerate-and-test approach. We start with the predicate we want to establish, compute the weakest precondition of the predicate with respect to the current statement, and finally perform a backward search through an inference system to find a valid antecedent that implies the weakest precondition. By performing a targeted search, our goal-directed approach prunes the search space very effectively, allowing us to generate rules for a total of 10 fact schemas in under 10 minutes on a modern desktop machine.

To give an overview of our technique, we describe the steps that our algorithm would take on a simple example. Consider the fact schema $hasConstValue$ on lines 1-2 of Figure 2. In this case, our problem of inferring propagation rules reduces to finding formulas ψ that make the propagation rule **if** ψ **then** $hasConstValue(X, C)@out$ correct. Intuitively, this rule is correct if, whenever ψ holds before $currStmt$, the meaning of $hasConstValue(X, C)$ holds after $currStmt$, i.e., ψ must be a precondition with respect to $currStmt$ that ensures $\sigma(X) = \sigma(C)$ as a postcondition. Since we wish to find the most general such ψ, our algorithm first computes the *weakest* precondition ϕ with respect to $currStmt$ that establishes $\sigma(X) = \sigma(C)$. As we shall see, the weakest precondition may refer to run-time information not available at analysis-time, and so ϕ itself may not be directly usable in a Rhodium propagation rule. Consequently, our algorithm searches through the space of formulas that imply ϕ to find formulas ψ that refer only to analysis-time information (i.e., dataflow fact instances and syntactic tests of statement forms). In logical terms, the process of finding formulas ψ that imply ϕ is called *strengthening*.

To compute the weakest precondition of $\sigma(X) = \sigma(C)$ with respect to $currStmt$, we do a case analysis on the possible forms of $currStmt$. Consider for example the case where $currStmt$ is of the form $Y := K$, where Y and K range respectively over variables and constants of the IL program. The traditional weakest precondition rule for an arbitrary assignment is as follows:

$$wp(Z := E, P) = P[Z \mapsto E]$$

where $P[Z \mapsto E]$ denotes the formula P with Z replaced by E.[1] In our case, we would therefore have:

$$\begin{aligned} \phi &= wp(Y := K, \sigma(X) = \sigma(C)) \\ &= (\sigma(X) = \sigma(C))[Y \mapsto K] \end{aligned}$$

In the above equation, the Rhodium meta-variables X and Y range over IL variables. The result of the substitution operation $[Y \mapsto K]$ thus depends on whether or not X and Y are instantiated to the same IL variable. If they are, then $(\sigma(X) = \sigma(C))[Y \mapsto K]$ expands to $\sigma(K) = \sigma(C)$; otherwise, it remains $\sigma(X) = \sigma(C)$. Since K and C are constants, $\sigma(K) = \sigma(C)$ in turn simplifies to $K \triangleq C$ (in the rest of this paper, we use the symbol $\triangleq$ for syntactic equality of abstract syntax trees (including variable names and constants) and $=$ for semantic equality of run-time values). The weakest precondition ϕ is then as follows:

$$\phi = (X \triangleq Y \wedge K \triangleq C) \vee (X \not\triangleq Y \wedge \sigma(X) = \sigma(C))$$

The antecedent of a rule cannot contain references to σ since σ is only known at run-time, not analysis-time. As a result, we cannot use the above weakest precondition directly as the antecedent ψ. However, the antecedent of a rule can refer to dataflow facts, and the meanings of these dataflow facts can refer to σ. Our task then is to "encapsulate" all the σ's in ϕ using dataflow facts. In the constant propagation example, the user would provide the dataflow fact schema $hasConstValue(X, C)$ with meaning $\sigma(X) = \sigma(C)$. By searching for syntactic matches of the form $\sigma(X) = \sigma(C)$ in ϕ and replacing them with $hasConstValue(X, C)@in$, we get:

$$\psi = (X \triangleq Y \wedge K \triangleq C) \vee (X \not\triangleq Y \wedge hasConstValue(X, C)@in)$$

This antecedent does not refer to σ and can therefore be used to generate a correct propagation rule for statements of the form $Y := K$:

if $currStmt \triangleq [Y := K] \wedge \psi$
then $hasConstValue(X, C)@out$

If we wish, we can split the disjunction in ψ into two separate rules:

if $currStmt \triangleq [Y := K] \wedge X \triangleq Y \wedge K \triangleq C$
then $hasConstValue(X, C)@out$

if $currStmt = [Y := K] \wedge X \not\triangleq Y \wedge hasConstValue(X, C)@in$
then $hasConstValue(X, C)@out$

The rules above are equivalent to the handwritten rules from lines 4-5 and 6-7 of Figure 2.

In this simple illustrative example, the mapping from the weakest precondition ϕ to a valid antecedent ψ was immediate. In more complicated cases, the mapping will not be immediate. Our algorithm will perform logical rewrites in an attempt to find a way to eliminate all σ's. Because we are searching for formulas that imply the weakest precondition ϕ, permissible rewrites include simplifications (finding a ϕ' that is equivalent to ϕ) and strengthenings (finding a less general ϕ' that implies ϕ). Intuitively, strengthenings sacrifice precision to make the condition ϕ statically computable in terms of available facts.

Each logical rewrite, be it a simplification or a strengthening, can be seen as a single backward step in an inference system. Our algorithm therefore performs a backward search through an inference system starting at the weakest precondition ϕ. All of the formulas considered in this backward search will imply ϕ, and our goal is to find such a formula that does not contain references to σ. Once we find such a ϕ, the sequence of inferences we performed during the search, if reversed to be in the forward direction, will constitute a proof of correctness for the rule we just generated. Indeed, this forward sequence of inference steps is a derivation of the condition that our Rhodium checker would send to the theorem prover on the newly generated rule. In this way, the rules we generate are guaranteed to be correct.

The search that we perform through an inference system is similar in many ways to searches that are done in automated or semi-automated theorem provers. Our search is goal-directed, in that we start with a goal, and search backwards through the space of proof-trees to find formulas that imply the goal. Many theorem provers use such a goal-directed search, for example PVS [19], NuPRL [4], Twelf [21, 28], the Boyer Moore theorem prover [12, 13], Isabelle [20], and HOL [8]. However, the key difference between our work and automated theorem proving lies in the problem that we are solving. A theorem prover is given a set of axioms, and asked to prove a theorem. If it uses a backward search, the theorem prover starts with the theorem to be proven and applies inference rules backward to reach the axioms. Our problem is different because we are *not* given axioms that we must search towards. The goal of our backward search is not to reach a set of axioms, but rather to reach a formula in a restricted form, i.e., one that no longer contains occurrences of σ. Although the goals are different, there are nonetheless strong similarities in the techniques. In particular, users of semi-automated theorem provers often use tactics (or variations thereof) to guide the theorem prover in its backward search of the large proof-tree space. One can view our work as a set of specialized tactics for the purposes of finding a statically computable formula from a formula that mentions run-time values.

4. Algorithm

Our algorithm for generating rules is shown in Figure 3. To simplify the presentation, we assume that statements have only one outgoing CFG edge called out – our implementation, however, also handles statements with multiple outgoing edges, such as conditional branches. The function $GenerateRules$ (lines 1-9) takes a set of fact schema declarations and returns a set of propagation rules. For each fact schema $F(\vec{V})$ declared by the user, for each statement form S (covering the different kinds of IL statements defined in Figure 1), the algorithm finds the rules that propagate fact $F(\vec{V})$ on the edge out of statement S. In the body of this loop, we first compute the weakest precondition of the meaning of fact $F(\vec{V})$ with respect to the statement form, using the wp function (line 4). We then perform a backward search in our inference system starting from the weakest precondition (line 5). The backward search is a loop that ends when we have removed all σ's from ϕ. The first step in the loop is to express ϕ in a simplified form using the $Simplify$ function (line 10). If all references to σ have been removed (line 11), then we have found a valid antecedent and the search is over (line 12). If not, we strengthen the formula using $Strengthen$ in order to remove some of the references to σ (line 13), and then we continue the search with the resulting stronger formula (line 14).

The algorithm in Figure 3 depends on the three functions wp, $Simplify$, and $Strengthen$. These three functions are described in more detail in the following three subsections.

Throughout our explanations, we will use the examples in Figures 4 and 5, which show the inference steps used by our algorithm for $hasConstValue(X, C)$ on statements of the forms $Y := K$ and $*Y := Z$, respectively. In these examples, the fact schemas available for matching are $hasConstValue(X, C)$ with meaning $\sigma(X) = \sigma(C)$, $mustNotPointTo(X, W)$ with meaning $\sigma(X) \neq \sigma(\&W)$, and $mustPointTo(X, W)$ with meaning $\sigma(X) = \sigma(\&W)$. The examples should be read bottom-up: the bottommost formula is the meaning of $hasConstValue(X, C)$, and the topmost formula is the final predicate ψ that we use to generate a rule. The final top-down sequence represents a valid logical

[1] This rule ignores the potential aliasing effects of pointers and meta-variables. We return to this issue in Section 4.1.

```
function GenerateRules(decls: set[FactSchemaDecl]): set[Rule]
1.    let results := ∅
2.    for each [define edge fact schema F(V⃗) with meaning M] ∈ decls do
3.        for each statement form S do
4.            let φ := wp(S, M)
5.            let ψ := RemoveRuntimeState(φ, decls)
6.            if ψ ≠ false then
7.                let rule := "if currStmt ≙ S ∧ ψ then F(V⃗)@out"
8.                results := results ∪ {rule}
9.    return results

function RemoveRuntimeState(φ: Formula, decls: set[FactSchemaDecl]): Formula
10.   let φ_simp := Simplify(φ)
11.   if φ_simp contains no σ then
12.       return φ_simp
13.   let φ_stren := Strengthen(φ_simp, decls)
14.   return RemoveRuntimeState(φ_stren, decls)
```

Figure 3. Algorithm for generating rules from fact declarations

$$
\begin{array}{rl}
(5) & \dfrac{(Y \triangleq X \wedge K \triangleq C) \ \vee \ (Y \not\triangleq X \wedge hasConstValue(X,C)@in)}{(\forall\sigma.[Y \triangleq X] \wedge \forall\sigma.[K \triangleq C]) \ \vee \ (\forall\sigma.[Y \not\triangleq X] \wedge \forall\sigma.[\sigma(X) = \sigma(C)])} & \textit{logSimp, fact match} \\
(4) & \dfrac{\forall\sigma.[Y \triangleq X \wedge K \triangleq C] \ \vee \forall\sigma.[Y \not\triangleq X \wedge \sigma(X) = \sigma(C)]}{\forall\sigma.[(Y \triangleq X \wedge K \triangleq C) \ \vee (Y \not\triangleq X \wedge \sigma(X) = \sigma(C))]} & \textit{logSimp} \\
(3) & \dfrac{\forall\sigma.[(\sigma(\&Y) = \sigma(\&X) \wedge \sigma(K) = \sigma(C)) \ \vee (\sigma(\&Y) \neq \sigma(\&X) \wedge \sigma(X) = \sigma(C))]}{\forall\sigma \in \Sigma_{out}.\,(\sigma(X) = \sigma(C))} & \textit{case} \\
(2) & & \triangleq_{\&}, \not\triangleq_{\&}, \triangleq_{c} \\
(1) & & wp(Y := K)
\end{array}
$$

Figure 4. Inference steps for $hasConstValue(X, C)$ for statements of the form $Y := K$

$$
\begin{array}{rl}
(5) & \dfrac{\begin{array}{c}(mustPointTo(Y,X)@in \wedge hasConstValue(Z,C)@in) \vee \\ (mustNotPointTo(Y,X)@in \wedge hasConstValue(X,C)@in) \vee \\ (hasConstValue(Z,C)@in \wedge hasConstValue(X,C)@in)\end{array}}{\begin{array}{c}(\forall\sigma.[\sigma(Y) = \sigma(\&X)] \wedge \forall\sigma.[\sigma(Z) = \sigma(C)]) \vee \\ (\forall\sigma.[\sigma(Y) \neq \sigma(\&X)] \wedge \forall\sigma.[\sigma(X) = \sigma(C)]) \vee \\ (\forall\sigma.[\sigma(Z) = \sigma(C)] \wedge \forall\sigma.[\sigma(X) = \sigma(C)])\end{array}} & fact\ match \\
(4) & \dfrac{\begin{array}{c}\forall\sigma.[\sigma(Y) = \sigma(\&X) \wedge \sigma(Z) = \sigma(C)] \vee \\ \forall\sigma.[\sigma(Y) \neq \sigma(\&X) \wedge \sigma(X) = \sigma(C)] \vee \\ \forall\sigma.[\sigma(Z) = \sigma(C) \wedge \sigma(X) = \sigma(C)]\end{array}}{\forall\sigma.[(\sigma(Y) = \sigma(\&X) \wedge \sigma(Z) = \sigma(C)) \ \vee (\sigma(Y) \neq \sigma(\&X) \wedge \sigma(X) = \sigma(C))]} & \textit{logSimp} \\
(3) & & \forall\ \textit{resolution} \\
(1) & & wp(*Y := Z)
\end{array}
$$

Figure 5. Inference steps for $hasConstValue(X, C)$ for statements of the form $*Y := Z$

deduction. Each backward inference step represents one or more steps taken by wp, $Simplify$, and/or $Strengthen$. The labels to the right of an inference step identify exactly what steps were taken. These labels refer to rewrite rules, except for wp, which refers to the application of the wp function. The collection of all rewrite rules in our system can be found in Figures 7 and 8

4.1 Weakest Precondition

The function $wp(S, P)$ computes the weakest liberal precondition of a predicate P with respect to a statement S. The weakest liberal precondition is the weakest condition Q such that if Q holds before the statement S, and S terminates, then P will hold after S. [2]

It will be important in our algorithm to make the quantification over σ explicit in all formulas. For example, the meaning for $hasConstValue(X, C)@Edge$ becomes $\forall\sigma \in \Sigma_{Edge}.\sigma(X) = \sigma(C)$, where Σ_{Edge} is the set of all possible states the program can be in if and when control reaches the edge $Edge$. (We will often omit "$\in \Sigma_{in}$" clauses.)

The traditional method for calculating $wp(S, P)$ directly manipulates the expressions in P based on the effect of S [10]. However, direct manipulation becomes increasingly complicated in the presence of aliasing, e.g., via pointers or via Rhodium metavariables that might or might not be instantiated to the same program variables. For example, consider computing the weakest pre-

[2] A *liberal* precondition guarantees the postcondition *only* under the assumption that the statement terminates without error, as opposed to a *strict* precondition, which must also ensure that the statement terminates without error. The liberal version is appropriate for our task because the meaning of any dataflow facts computed on a CFG edge must hold for all possible program states *if and when* control reaches that edge; analysis results need not ensure that control actually reaches any particular edge.

$$
\begin{array}{lcl}
step(decl\ X, (\rho, \eta)) & = & (\rho[X \mapsto newloc], \\
 & & \ \eta[newloc \mapsto \bot]) \\
step(X := C, (\rho, \eta)) & = & (\rho, \eta[\rho[X] \mapsto C]) \\
step(X := \&Y, (\rho, \eta)) & = & (\rho, \eta[\rho[X] \mapsto \rho[Y]]) \\
step(X := Y, (\rho, \eta)) & = & (\rho, \eta[\rho[X] \mapsto \eta[\rho[Y]]]) \\
step(X := *Y, (\rho, \eta)) & = & (\rho, \eta[\rho[X] \mapsto \eta[\eta[\rho[Y]]]]) \\
step(*X := Y, (\rho, \eta)) & = & (\rho, \eta[\eta[\rho[X]] \mapsto \eta[\rho[Y]]])
\end{array}
$$

The $newloc$ symbol stands for a fresh location. $\bot$ is a value that gives a run-time error if read.

Figure 6. Some cases of the $step$ function

condition of the formula $\forall\sigma \in \Sigma_{out}.\sigma(X) = \sigma(C)$ with respect to a pointer store statement $*Y := Z$, as in Figure 5. When run on a particular program, the X meta-variable might be instantiated to the same program variable as the Y and/or Z meta-variables, or to a variable pointed to by the program variable that Y is instantiated to. We must compute a correct precondition in all these cases.

To solve these problems, we use an approach based on the work by Cartwright and Oppen [3]. The key idea is that

$$wp(S, \forall\sigma_o \in \Sigma_{out}.P)$$

is equal to

$$\forall\sigma_i \in \Sigma_{in}.P[\sigma_o \mapsto step(S, \sigma_i)]$$

where the $step$ function represents the operational semantics of our IL in terms of standard select and update map operators applied to the environment ρ and store η components of the incoming program state σ_i. Some of the cases for the $step$ function are shown in Figure 6; our technical report [27] contains full details. Finally, selections from updated maps M are simplified using the following rule:

$$(M[X \mapsto V])[Y] = \text{if } X = Y \text{ then } V \text{ else } M[Y]$$

This simplification naturally introduces the necessary aliasing-condition case analysis.

For example, consider computing

$$wp(*Y := Z, \forall\sigma_o \in \Sigma_{out}.\sigma_o(X) = \sigma_o(C))$$

Replacing σ with explicit ρ_o and η_o and expanding $\sigma(\cdot)$ short-hands yields

$$wp(*Y := Z, \forall(\rho_o, \eta_o) \in \Sigma_{out}.\eta_o[\rho_o[X]] = C)$$

Applying the definition of wp gives

$$\forall(\rho_i, \eta_i) \in \Sigma_{in}. \\ (\eta_o[\rho_o[X]] = C)\,[(\rho_o, \eta_o) \mapsto step(*Y := Z, (\rho_i, \eta_i))]$$

Applying the definition of $step$ for $*Y := Z$ gives

$$\forall(\rho_i, \eta_i) \in \Sigma_{in}. \\ (\eta_o[\rho_o[X]] = C)\,[(\rho_o, \eta_o) \mapsto (\rho_i, \eta_i[\eta_i[\rho_i[Y]] \mapsto \eta_i[\rho_i[Z]]])]$$

Applying the substitution of output for input state components gives

$$\forall(\rho_i, \eta_i) \in \Sigma_{in}.(\eta_i[\eta_i[\rho_i[Y]] \mapsto \eta_i[\rho_i[Z]]])[\rho_i[X]] = C$$

Applying the select/update simplification gives

$$\forall(\rho_i, \eta_i) \in \Sigma_{in}. \\ (\text{if } \eta_i[\rho_i[Y]] = \rho_i[X] \text{ then } \eta_i[\rho_i[Z]] \text{ else } \eta_i[\rho_i[X]]) = C$$

Reexpressing using σ short-hands gives

$$\forall\sigma \in \Sigma_{in}.\,(\text{if } \sigma(Y) = \sigma(\&X) \text{ then } \sigma(Z) \text{ else } \sigma(X)) = \sigma(C)$$

which is equivalent to the expected weakest precondition

$$\forall\sigma \in \Sigma_{in}. \\ (\sigma(Y) = \sigma(\&X) \wedge \sigma(Z) = \sigma(C)) \vee \\ (\sigma(Y) \neq \sigma(\&X) \wedge \sigma(X) = \sigma(C))$$

Context Rules

$F^t[T_1] \rightsquigarrow F^t[T_2]$ if $T_1 \rightsquigarrow T_2$
$F[F_1] \rightsquigarrow F[F_2]$ if $F_1 \rightsquigarrow F_2$

Logical Simplifications

$[true_{elim}]$	$P \wedge true \rightsquigarrow P$
$[\neg_{push}]$	$\neg(P \wedge Q) \rightsquigarrow \neg P \vee \neg Q$
...many more...	

IL Simplifications

Term Rewrite Rules

$[\&*_E]$	$\sigma(\&(*X)) \rightsquigarrow \sigma(X)$
$[*\&_E]$	$\sigma(*(\&X)) \rightsquigarrow \sigma(X)$

Formula Rewrite Rules

$[\stackrel{\circ}{=}_{\&}]$	$\sigma(\&X) = \sigma(\&Y) \rightsquigarrow X \stackrel{\circ}{=} Y$
$[\stackrel{\circ}{=}_c]$	$\sigma(K) = \sigma(C) \rightsquigarrow K \stackrel{\circ}{=} C$
$[\mathsf{F}_{c\&}]$	$\sigma(C) = \sigma(\&X) \rightsquigarrow false$
$[\mathsf{F}_{\&\ op}]$	$\sigma(\&X) = \sigma(\ op\ T_1\ \ldots\ T_n) \rightsquigarrow false$
$[\mathsf{T}_{=}]$	$\sigma(T) = \sigma(T) \rightsquigarrow true$
$[\mathsf{T}_{\stackrel{\circ}{=}}]$	$T \stackrel{\circ}{=} T \rightsquigarrow true$
$[\mathsf{F}_{\stackrel{\circ}{=}}]$	$T \stackrel{\circ}{=} T' \rightsquigarrow false$ (if T and T' are incompatible forms)

Figure 7. Some simplification rules

4.2 Simplification

The *Simplify* function transforms a given formula into an equivalent formula that is in a simpler form by applying simplification rules until no more are applicable. This process involves two kinds of simplifications: logical simplifications, which simplify logical connectives; and IL simplifications, which simplify IL terms and expressions based on the semantics of our IL. IL simplifications include such simplifications as rewriting $*(\&Y)$ to Y.

We express the simplification process as a rewrite system, where the rewrite rule $F_1 \rightsquigarrow F_2$ says that the term or formula F_1 is rewritten to F_2. Figure 7 shows the most important of the rules of this rewrite system (the rest can be found in our technical report [27]). We use $F^t[\cdot]$ to represent a context F^t with a term hole and $F[\cdot]$ to represent a context F with a formula hole. The *context rules* allow terms and subformulas to be rewritten inside of a formula. *Logical simplifications* are used to put the formula in a simplified logical form. *IL simplifications* are used to simplify the formula based on the semantics of our IL; each of the formula rewrite rules also has a disequality version which is not shown explicitly.

4.3 Strengthening

If *Simplify* cannot remove all occurrences of σ from ϕ, the formula must be strengthened using the *Strengthen* function. Unlike the *Simplify* function, the *Strengthen* function makes a formula less precise. Our algorithm uses many strengthening rewrites; Figure 8 shows the most important of these. $F_1 \rightsquigarrow_S F_2$ denotes that F_1 can be rewritten to F_2 in a positive position, meaning that, from a logical point of view, $F_2 \Rightarrow F_1$. We use $F^+[\cdot]$ to represent a context F^+ with a formula hole that appears in a positive position in F^+. The context rule at the top of Figure 8 allows subformulas to be rewritten inside a formula. (Allowing strengthenings in negative holes would lead to a weaker overall formula, which is incorrect for our application. It would be safe to allow weakenings in negative holes.)

The *Strengthen* function collects all rewrites that apply to ϕ and applies each rewrite r separately to ϕ to produce ϕ_r. *Strengthen* then returns the disjunction of all the resulting ϕ_r formulas. Although in theory this causes a worst-case exponential run-

Context Rule

$$F^+[F_1] \rightsquigarrow_S F^+[F_2] \text{ if } F_1 \rightsquigarrow_S F_2$$

Strengthening Rules

$[fact\ match]$	for each [**define edge fact schema** $F(\vec{V})$ **with meaning** M] $\in decls$: $\quad \forall\sigma \in \Sigma.M \rightsquigarrow_S F(\vec{V})@in$
$[syntactic]$	$\sigma(T_1) = \sigma(T_2) \rightsquigarrow_S T_1 \triangleq T_2$
$[case]$	$\forall x.[F_1 \vee F_2] \rightsquigarrow_S \forall x.[F_1] \vee \forall x.[F_2]$
$[\forall\ resolution]$	$\forall x.[(F_1 \wedge F_2) \vee (\neg F_1 \wedge F_3)] \rightsquigarrow_S (\forall x.[F_1 \wedge F_2]) \vee (\forall x.[\neg F_1 \wedge F_3]) \vee (\forall x.[F_2 \wedge F_3])$
$[op_{expand}]$	$F^t[\sigma(\ op\ \ T_1 \ldots T_n)] \rightsquigarrow_S (\bigwedge_{i=1\ldots n} \sigma(C_i) = \sigma(T_i)) \wedge F^t[eval(\ op\ \ C_1 \ldots C_n)]$
$[quant\ swap]$	$\forall x.\exists y.P(x,y) \rightsquigarrow_S \exists y.\forall x.P(x,y)$
$[mathematical]$	Strengthening rewrites based on mathematical properties such as $\sigma(X) \leq \sigma(Y) \rightsquigarrow_S \sigma(X) < \sigma(Y)$

Figure 8. Strengthening rules

time, in practice only a small number of strengthenings apply to a given ϕ, and in our experience we have not seen the exponential worst case.

The strengthening rewrites can be divided into three categories: rewrites for introducing facts, rewrites for approximating run-time knowledge at analysis-time, and rewrites for encoding mathematical reasoning. The following three subsections describe each one of these in more detail.

Fact Matching. The first strengthening, labeled [*fact match*] , replaces a subformula of ϕ with an instance of a fact schema. In Figure 8, *decls* is the set of declared fact schemas that is passed in as a parameter to $Strengthen$. For each fact schema, our strengthening algorithm considers rewriting instances of the schema's meaning to instances of the fact.

Because we have made quantifiers explicit, the meaning of a fact is universally quantified over σ, so that, for example, the meaning of $hasConstValue(X, C)@in$ is $\forall\sigma \in \Sigma_{in}.[\sigma(X) = \sigma(C)]$. As a result, in step (5) of Figure 4, the $Strengthen$ function replaces the subformula $\forall\sigma.[\sigma(X) = \sigma(C)]$ with $hasConstValue(X, C)@in$, as indicated by the $[\forall\sigma.M \rightsquigarrow_S F(\vec{V})@in]$ label to the right of the inference step.

Mapping meanings to facts is a strengthening rather than an equivalence because the presence of a fact on an edge implies that its meaning holds at that edge, but not necessarily the reverse. The meaning might hold at an edge, but the fact might not have been computed on that edge, perhaps because the rest of the analysis was not precise enough to compute the fact whenever its meaning was true. Consequently, the absence of a Rhodium fact provides no information, and it would be unsound in general to map meanings to facts in negative positions.

Approximating Run-time Knowledge. The second kind of strengthening embodies the idea of approximating run-time knowledge with analysis-time knowledge. For example, the run-time predicate $\sigma(X) = \sigma(Y)$ is always true when the Rhodium meta-variables X and Y are instantiated to the same IL variable. Thus, the analysis-time information $X \triangleq Y$ approximates the knowledge that X and Y produce the same value at run time. The $[syntactic]$ rule captures the general case for arbitrary terms.

Another example of approximating run-time information is encoded in the $[case]$ rule, which approximates run-time choice with analysis-time choice. Intuitively, $\forall\sigma.[F_1 \vee F_2]$ allows each run-time σ to make an independent choice of whether F_1 or F_2 is true, while $\forall\sigma.[F_1] \vee \forall\sigma.[F_2]$ requires the analysis to choose whether F_1 or F_2 is true for every run-time σ. The $[\forall\ resolution]$ rule is a refinement of the $[case]$ rule that retains more precision when the F_i have a special form. The use of these case analysis rewrites has allowed our algorithm to infer useful rules that we had not thought of previously when writing the flow functions by hand.

Mathematical Reasoning. The third kind of strengthening encodes mathematical knowledge. One example of such a strengthening, shown in Figure 8, is $\sigma(X) \leq \sigma(Y) \rightsquigarrow_S \sigma(X) < \sigma(Y)$. The mathematical strengthenings used in our algorithm encode simple information about arithmetic. These mathematical strengthenings can easily be extended without changing the rest of the algorithm.

4.4 Termination

Theorem 1. $GenerateRules$ terminates on finite input.

Proof. A sketch of the proof is included here; the full proof can be found in our technical report [27]. The key question is whether the recursive $RemoveRuntimeState$ function terminates. Termination of this function is proven using strong induction over the height of the formula. For each of the rewrites potentially applied during simplification and strengthening, a case analysis over the possible forms of the formula being rewritten shows that only a bounded number of rewrites can follow, excluding rewrites on strict subformulas of the original formula, which inductively are assumed to terminate. □

5. Evaluation

To evaluate the effectiveness of our algorithm for automatically inferring propagation rules from fact schemas, we first defined a set of forward fact schemas and then wrote a collection of forward propagation and transformation rules by hand. These rules were intended to be as complete as possible, and represent a kind of "gold standard" against which other versions of the rules could be compared. These facts and rules compute information sufficient for a variety of traditional intraprocedural analyses and optimizations, including constant propagation, copy propagation, must- and may-point-to analyses for pointers to variables and to array elements, pointer-target-contents analysis suitable for scalar replacement, available expressions, an analysis tracking which variables point to stack-allocated local variables as opposed to heap-allocated data, dynamic type analyses tracking which variables hold ints, arrays, pointers, and pointers to array elements, strength reduction for induction variables, and symbolic range analysis. In total, this "gold standard" is comprised of 10 fact schemas, 116 propagation rules, and 15 transformation rules.

We then ran our tool on the same 10 fact schemas to generate propagation rules automatically. Our tool generated 6,185 propagation rules, taking just under 10 minutes on a modern workstation. Our tool generates a separate rule for each statement form. In con-

Category	Number	Percent
Covered	472	77.0
Introduces new variable or constant	77	12.6
Weakness of purely syntactic reasoning	56	9.1
Lost information	4	0.7
Missing strengthening rewrites	2	0.3
Missing mathematical reasoning	2	0.3
Total	613	100.0

Table 1. Coverage of handwritten rules by inferred rules

trast, some handwritten rules apply to several statement forms. If such handwritten rules are duplicated for each different statement form to which they apply, then the number of per-statement-form handwritten rules is 613, which is more directly comparable to the 6,185 automatically generated per-statement-form rules.

We wish to know how many of the facts computed by the handwritten rules would also be computed by the automatically generated rules, and whether there are any facts computed by the automatically generated rules that would have been missed by the handwritten rules. We compare the two sets of rules in two ways:

- In section 5.1, we perform a theoretical comparison to see if there could exist situations in which one set of propagation rules computes a fact while the other set does not.
- In section 5.2, we assess how much optimization is triggered by the two competing sets of propagation rules when optimizing a collection of benchmark programs.

5.1 Comparison in Principle

5.1.1 Methodology

To evaluate the theoretical quality of the rules inferred by our algorithm, we have developed a general method for comparing two sets of Rhodium rules without regard to any particular program that the rules may run on. If we let S_g be the set of automatically generated rules, and S_h of be the set of handwritten rules, then we want to know which of the rules in S_h are covered by rules in S_g, and vice versa. A rule r propagating an instance of a fact schema f is *covered* by a set of rules S if, whenever r fires (i.e., whenever its antecedent holds) and propagates a particular instance of f, some rule from S would also fire and propagate the same instance of f. A rule $r =$ **if** ψ **then** $f(\vec{T})@out$ is covered by the rules in $S = \{s_1, \ldots, s_m\}$ iff $\psi \wedge |s_1| \wedge \ldots \wedge |s_m| \Rightarrow f(\vec{T})@out$ where $|s_i|$ is the translation of a rule s_i into closed logical form:

$$|\textbf{if } \psi_i \textbf{ then } f_i(\vec{T_i})@out| = \forall X_1, \ldots, X_n \,.\, \psi_i \Rightarrow f_i(\vec{T_i})@out$$

and $X_1, \ldots, X_n$ are the free variables of s_i. This condition, along with axioms describing the semantics of our IL, is sent to the Simplify theorem prover [7]. If the theorem prover returns valid, then we know that S covers r, i.e., that whenever r would compute a fact instance, S would too. If the theorem prover cannot show the above condition, then we gain no information (since the theorem prover is incomplete). If the theorem prover can show the negation of this condition, then we know that the S does *not* cover r.

By testing coverage of each rule $r \in S_h$ against $S = S_g$, we can determine which of the handwritten rules are covered by the automatically generated ones. Conversely, by testing coverage of each rule $r \in S_g$ against $S = S_h$, we can find automatically generated rules that are *not* covered by the handwritten ones, i.e., that are *novel*. In the following two subsections, we present comparison results for both of these directions.

5.1.2 Comparing Handwritten Rules to Inferred Rules

We first applied our comparison algorithm to determine what rules from our handwritten "gold standard" are covered by the automatically generated rules. The results of this comparison are shown in table 1. The first line of the table says that our inference method is able to infer rules that cover 472 (77%) of the 613 expanded handwritten rules. The rest of the table reports the number of handwritten rules that were not covered by any set of inferred rules, broken down into categories. Overcoming these limitations is an important area for future work.

- **Introduces new variable or constant:** Our inference algorithm only infers rules containing meta-variables defined in the statement form or the propagated fact. Some handwritten rules, particularly those performing a kind of transitive closure operation over facts, mention additional "intermediate" meta-variables.
- **Weakness of purely syntactic reasoning:** A rewrite in our system is triggered only if its left-hand-side exactly matches some subformula of the current ϕ. This syntactic matching algorithm leads to cases where superficial changes in a formula or meaning affects whether or not the algorithm can detect a match. Just as one example, there are cases where the resolution strengthening rewrite $[\forall\ resolution]$ is not applied because the current formula does not exactly match the form of the left-hand-side of $[\forall\ resolution]$, even though it is logically equivalent to the left-hand-side.
- **Lost information:** There is some information about the execution of IL statements that we have not yet added to our inference algorithm. For example, our algorithm is not aware of the fact that after a branch statement, the guard is true on $out[0]$ and false on $out[1]$. As a result, handwritten rules that exploit this information are not covered. Another kind of unused information is dynamic typing. For example, after the statement `decl` $X[I]$, X is guaranteed to be an array. Handwritten rules that make use of this information are not covered.
- **Missing strengthening rewrites:** Our inference algorithm does not currently detect whether or not it has been in a particular state before, and to ensure termination, the set of strengthening rewrites cannot include rewrites that may be potentially cyclic. For example, we can only include one of the two following sets of strengthening rules:

  $$\sigma(T_1) = \sigma(T_2) \rightsquigarrow_S \\ \quad \sigma(T_1 \leq T_2) = \sigma(true) \wedge \sigma(T_1 \geq T_2) = \sigma(true)$$

  or

  $$\sigma(T_1 \leq T_2) = \sigma(true) \rightsquigarrow_S \sigma(T_1) = \sigma(T_2) \\ \sigma(T_1 \geq T_2) = \sigma(true) \rightsquigarrow_S \sigma(T_1) = \sigma(T_2)$$

  We could remove this limitation by adding a kind of closure operation that performs a collection of related strengthenings all at once, and never again.
- **Missing mathematical reasoning:** Some of the handwritten rules exploit mathematical reasoning, for example distributivity and commutativity, that is currently not encoded in our rules for simplification and strengthening. One approach to solving this problem would be to add rules that encode these mathematical concepts. Alternatively, one could investigate ways of integrating decision procedures that perform mathematical reasoning into the inference algorithm.

5.1.3 Comparing Inferred Rules to Handwritten Rules

We also used our comparison method to determine how many automatically generated rules were covered by the handwritten rules. Table 2 summarizes the results. In particular, of the 6,185 automatically generated rules, our comparison method finds that 3,697 rules were not shown to be covered, and so may be novel.

Category	Number	Percent
Covered	2,488	40.2
Potentially novel	3,697	59.8
Total	6,185	100.0

Table 2. Coverage of inferred rules by handwritten rules

Benchmark	Lines of IL	Handwritten only	Both	Inferred only
test suite	7,432	32	578	57
Java stdlib	55,993	41	847	0
soturon	396	0	16	0
counter	599	0	24	0
towers	909	0	57	0
mandelbrot	1,708	0	59	0
cassowary	14,747	38	352	0
raytrace	16,836	0	320	0
esspresso	81,355	43	1,517	2
Total	–	154	3,770	59

Table 3. Number of triggered transformations

There are too many potentially novel rules to systematically analyze manually. Instead, we will give an example of an inferred rule that is novel:

if $currStmt \triangleq [*Z := Y]$
$\wedge\ varEqualsExpr(X, W)@in \wedge varEqualsExpr(Y, W)@in$
$\wedge\ mustNotPointTo(Z, W)@in$
then $varEqualsExpr(X, W)@out$

This rule says that if X and Y are both equal to W before $*Z := Y$, and the assignment does not change W, then X will be equal to W after the assignment. As first, this rule may seem surprising, because it propagates $varEqualsExpr(X, W)$ unchanged through a pointer store $*Z := Y$ without knowing whether or not Z points to X. This rule is nonetheless correct: if Z points to X, then the assignment is storing Y in X, in which case $varEqualsExpr(Y, W)@in$ guarantees $varEqualsExpr(X, W)@out$; on the other hand, if Z does *not* point to X, then the assignment does not modify X, in which case $varEqualsExpr(X, W)@in$ guarantees $varEqualsExpr(X, W)@out$. Although this is a syntactically simple rule, the effects of pointers and mutation make it difficult to reason about. This example, along with the large number of potentially novel rules, points out that humans are not good at finding all the corner cases for dataflow analyses; our inference algorithm helps to cover such cases.

5.2 Comparison in Practice

The theoretical comparison from section 5.1 showed that the rules inferred by our algorithm cover a reasonably large proportion of the handwritten rules. However, the theoretical comparison does not indicate how useful the rules are in practice. We would like to know whether the rules inferred by our algorithm trigger as many program transformations as the handwritten rules. Using the Rhodium execution engine in Whirlwind, we optimized several benchmark programs, once with the handwritten rules and once with the inferred rules. Table 3 lists the benchmarks and gives the number of lines of Whirlwind IL in each. "test suite" is a handwritten regression test suite, while the others are translated mechanically from Java .class files. "Java stdlib" is the translation of 135 Java classes from the core of the Java standard library, while the other seven are the translation of small- to medium-sized Java programs, excluding the standard library.

We counted the number of times any of the 15 transformation rules in our "gold standard" was triggered by the handwritten rules and the inferred rules. Table 3 shows the results of our comparison. The column labeled "Handwritten only" shows the number of transformations triggered by the handwritten rules, but not by the inferred ones; the column labeled "Inferred only" shows the number of transformations triggered by the inferred rules, but not by the handwritten ones; and the column labeled "Both" shows the number of transformations triggered by both sets of rules.

Summing across all benchmarks, the inferred rules triggered over 95% of the optimizing transformations triggered by the handwritten rules, and some missed by the handwritten rules.

6. Related Work

As pointed out in section 3, our work is a solution to a generalized version of the predicate abstraction problem called parametric predicate abstraction. One of the limitations of predicate abstraction that this generalization addresses is the finite-domain restriction in predicate abstraction.

The recent work of Reps, Sagiv, and Yorsh [25, 29] also addresses the finite-domain limitation of the predicate-abstraction approach: they have derived the best flow function for a more general class of domains, namely finite-height domains. For these domains, Reps *et al.* present an algorithm that computes the best possible abstract information flowing out of a statement given the abstract information flowing into it. Their algorithm has the nice theoretical property of providing the best possible transformer, a property which we do not guarantee. However, the flow function of Reps *et al.* is not specialized with respect to the domain: they describe one single flow function that works for all finite-height domains. As a result, each invocation of the flow function uses an iterative approximation technique that makes successive calls to a decision procedure (a theorem prover). In contrast, our approach generates flow functions that are specific to the domain specified by the user, and as a result our generated flow functions can be expressed as simple rules that only perform syntactic checks. Another way to view the difference between our work and that of Reps *et al.* is that we try to pre-compute as much as possible of the flow functions when we generate them, leaving little work for when the flow functions are executed, whereas Reps *et al.* do all the work when the flow functions are executed. Finally, the approach that we take is different in nature from the Reps *et al.* approach. Our algorithm is goal-directed, in that we start with the fact that we want to propagate after the statement, and then work our way backwards to the condition that must hold before the statement. In contrast, the Reps *et al.* approach works in the forward direction.

Another system that infers flow functions automatically is the TVLA system [18]. TVLA can automatically generate the abstract semantics of a program from its concrete semantics. Here again, the main difference compared to our work is that the original TVLA system runs an interpretation algorithm in the forward direction on every call to the flow function. Our system, on the other hand, precomputes much of the flow function when generating it, resulting in simple flow functions that can be evaluated using syntactic checks.

Reps *et al.* have extended the original TVLA system with an approach for automatically computing transfer functions for TVLA instrumentation predicates [24]. In this extended TVLA system, the transfer functions are computed ahead of time, before the analysis even runs once, as in our work. However, the Reps *et al.* approach is different from ours in a variety of ways. Their technique is based on a finite-differencing characterization of the relationship between the predicates holding before and after a given statement. In contrast, our approach uses a standard weakest precondition computation to capture this relationship, but we introduce a variety of strengthenings in order to re-express the weakest precondition in terms of a given set of fact schemas. Also, the Reps *et al.* approach focuses on the three-valued logic domain and pointer analysis, and

it currently cannot directly handle constants and function symbols such as plus and minus.

Our work is also related to the HOIST system for automatically deriving static analyzers for embedded systems [23, 22]. The HOIST work derives abstract operations by creating a table of the complete input-output behavior of concrete operations, and then abstracting this table to the abstract domain. Our work differs from HOIST in that we can handle concrete domains of infinite size, whereas the HOIST approach inherently requires the concrete domain to be finite.

As described in section 3, the search that we perform through an inference system is closely related to the ideas of tactics and tacticals from automated theorem provers. Another related proof-search technique is focusing [1], which is a way of alternating the application of so-called invertible rules (rules where the premise is equivalent to the conclusion) and non-invertible rules (rules where the premise implies but is not equivalent to the conclusion). In particular, invertible rules are applied eagerly until none apply, and then non-invertible rules are repeatedly applied to a *focused* subformula until invertible rules again become applicable. As with focusing, we exhaustively apply invertible rules (during our simplification phase), and then we apply non-invertible rules (during our strengthening phase) to uncover more opportunities for applying invertible rules.

A preliminary design for our inference algorithm was presented informally at the COCV workshop [26].

7. Conclusion

We have presented an algorithm for automatically inferring dataflow analysis flow functions from dataflow fact schemas that are ascribed semantic meanings. By reducing the burden on the analysis-writer, while at the same time guaranteeing that flow functions are correct, we hope that our work will not only make it easier to write correct program analysis tools, but will also make it feasible to open up program analysis tools to safe user extension.

In the future, we would like to pursue less-syntactic (and therefore less-brittle) methods for representing and manipulating formulas, such as using the E-graph data structure from Simplify [7] to better represent equalities. We also would like to investigate inferring not only propagation rules but also the fact schemas themselves from just a set of desirable transformations.

References

[1] J.M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 1992.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[3] Robert Cartwright and Derek Oppen. The logic of aliasing. Technical Report STAN-CS-79-740, Stanford University, September 1979.

[4] 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, 1986.

[5] Patrick Cousot. Verification by abstract interpretation. In *Verification – Theory & Practice*, volume 2772 of *LNCS*. Springer-Verlag, 2003.

[6] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification (CAV)*, June 1999.

[7] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3), May 2005.

[8] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In *VLSI Specification Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[9] Susanne Graf and Hassen Saidi. Construction of abstract state graphs of infinite systems with PVS. In *Computer Aided Verification (CAV)*, June 1997.

[10] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[11] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, January 2002.

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

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

[14] Shuvendu Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. In *Computer Aided Verification (CAV)*, July 2005.

[15] Sorin Lerner. *Automatically Proving the Correctness of Program Analyses and Transformations*. PhD thesis, University of Washington, March 2006.

[16] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[17] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Principles of Programming Languages (POPL)*, January 2005.

[18] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, June 2000.

[19] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction (CADE)*, volume 607 of *LNAI*. Springer-Verlag, 1992.

[20] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[21] F. Pfenning and C. Schurmann. Sytsem description: Twelf – a meta-logical framework for deductive systems. In *Automated Deduction (CADE)*, volume 1632 of *LNAI*. Springer-Verlag, July 1999.

[22] John Regehr and Usit Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2006.

[23] John Regehr and Alastair Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[24] Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming (ESOP)*, April 2003.

[25] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2004.

[26] Erika Rice, Sorin Lerner, and Craig Chambers. Automatically inferring sound dataflow functions from dataflow fact schemas. In *Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, April 2005.

[27] Erika Rice, Sorin Lerner, and Craig Chambers. Automatic inference of dataflow analyses. Technical Report UW-CSE-2006-06-04, University of Washington, June 2006.

[28] C. Schurmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Automated Deduction (CADE)*, volume 1421 of *LNCS*. Springer-Verlag, July 1998.

[29] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, March 2004.