Automatic Inference of Dataflow Analyses

University of Washington TR 2006-06-04

Erika Rice

June 9, 2006

Abstract

Much of the work of defining compiler optimizations is in writing dataflow analysis flow functions. We leverage some properties of the Rhodium optimization language to automatically infer sound dataflow analyses. Our technique infers 77% of the rules written by hand by an expert and infers many rules that cover cases omitted in the handwritten rules.

1 Introduction

Much of the work of defining compiler optimizations is in writing dataflow analysis flow functions. These rules define how to annotate a program with information that is useful for performing optimizations. For example, our implementation of 10 dataflow analyses includes 116 rules defining flow functions. Most rules are simple, but determining whether the rules cover all possible cases is difficult.

Our goal is to automatically infer dataflow analysis flow functions in Rhodium, a domain-specific language for writing compiler optimizations. Rhodium is supported by a verifier that can automatically prove code transformations and dataflow analysis flow function sound. Our inference engine leverages two properties of Rhodium. First, Rhodium expresses flow functions and code transformations by a set of declarative rules. The restricted form of the flow functions makes them easier to infer. Second, Rhodium requires optimization writers to supply a semantic meaning for the information computed by the flow functions. The meaning is a first-order logic formula which can be used as a starting point for inference. Together, this meaning and a name define a dataflow fact. A dataflow fact and a set of dataflow analysis flow functions make up a dataflow analysis.

Our inference technique starts with a dataflow fact and finds a set of dataflow analysis flow functions that define a sound but possibly incomplete dataflow analysis for that dataflow fact. Our technique infers 77% of the rules written by hand by an expert. It also infers rules omitted in the handwritten rules. Because we are lacking an efficient execution engine for Rhodium, we cannot test how useful these rules are in practice.

The rest of this paper is organized as follows. Section 2 introduces Rhodium. Section 3 gives an overview of our inference algorithm. Section 4 details our inference algorithm. Section 5 presents our implementation of the algorithm. Section 6 compares the rules our algorithm infers to a set of handwritten rules. Section 7 discusses future work and section 8 describes related work. Finally, we conclude in section 9.

2 Background on Rhodium

Rhodium is a domain-specific language for writing provably correctly optimizations and dataflow analyses. Rhodium analyses run over a C-like intermediate language (IL) with functions, recursion, pointers to dynamically allocated memory and to local variables, and arrays. For the purposes of our work on inferring analyses, we will consider a simpler IL without function calls, shown in Figure 1. The IL program is represented as a control flow graph (CFG) with each node representing a simple statement.

Stmts	s	$::=$	`skip` $\mid$ `decl` x $\mid$ `decl` $x[x]$ $\mid$ $x :=$ `new`
			$\mid$ $x :=$ `new`$[x]$ $\mid$ $x := e$ $\mid$ $*x := x$
			$\mid$ `if` x `goto` ι `else` ι
Exprs	e	$::=$	$c \mid x \mid *x \mid \&x \mid x \;\; op \;\; x$
			$\mid x[x]$
Ops	op	$::=$	$+ \mid - \mid \times \mid < \mid > \mid \leq \mid \geq$
ILVars	x	$::=$	`x` $\mid$ `y` $\mid$ `z` $\mid \ldots$
Consts	c	$::=$	constants
Labels	ι	$::=$	node identifiers

Figure 1: Grammar of the IL

PropRule	r	$::=$	`if` ψ `then`f
Antecedents	ψ	$::=$	$f \mid t \mathrel{\hat{=}} t \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi$
			$\mid$ `forall` $decl.\psi$ $\mid$ `exists` $decl.\psi$
VarDecl	$decl$	$::=$	$id : \tau$
Identifiers	id	$::=$	string literals
Types	τ	$::=$	`Const` $\mid$ `Var` $\mid$ `Expr`
EdgePred	f	$::=$	$id(t/,)@ed$
Terms	t	$::=$	$id \mid [e] \mid$ `currStmt` $\mid [s]$
Edge	ed	$::=$	`in` $\mid$ `out` $\mid$ `in`$[c]$ $\mid$ `out`$[c]$
Exprs	e	$::=$	exprs over IL vars and metavars
Stmts	s	$::=$	statements over IL vars and metavars
Metavars	X	$::=$	$X \mid Y \mid Z \mid \ldots$

Figure 2: Grammar of Rhodium dataflow analysis propagation rules

```
1. decl $X$:Var, $C$:Const

2. define edge fact schema hasConstValue($X$:Var, $C$:Const)
3. with meaning $\sigma(X) = \sigma(C)$

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

6. if hasConstValue$(X, C)$@in $\wedge$ currStmt $\doteq [Z := K] \wedge X \not\doteq Z$
7. then hasConstValue$(X, C)$@out
```

Figure 3: Simple constant propagation analysis in Rhodium.

Dataflow information is encoded in Rhodium with dataflow facts; these are user-defined function symbols applied to a set of terms. Two examples are *hasConstValue*(x, 5) and *exprIsAvailable*(x, a + b). A Rhodium analysis computes a set of dataflow facts at each edge between nodes in the CFG. A Rhodium analysis uses *propagation rules*, which are a stylized way of writing analysis flow functions, to specify how dataflow facts are generated or propagated across CFG nodes. These user-defined flow functions define a dataflow analysis.

The example in Figure 3 shows a partial Rhodium implementation of constant propagation which computes when a variable must be equal to a known constant. We encode the dataflow information for this analysis using the *hasConstValue*(X, C) *edge fact schema* declared on line 2. An edge fact schema is a parameterized dataflow fact (a pattern, in essence) that can be instantiated to create actual dataflow facts. Each edge in the CFG may be annotated with a set containing facts of the form *hasConstValue*(X, C) where X is instantiated with a particular program variable (e.g., x) and C is instantiated with an actual constant (e.g., 7).

To verify Rhodium flow functions and code transformations, programmers must specify a semantic meaning for each fact schema. The meaning is a first-order logic predicate over a program state, σ. In the user provided meaning M the program state σ is a free variable. The meaning is represented internally as $\forall\sigma \in \Sigma_{ed}.M$; the internal representation makes explicit that the meaning of a dataflow fact must hold for all program states that can appear on a CFG edge, *ed*, annotated with a dataflow fact. From now on, when we refer to the meaning of a fact, we refer to its internal representation. The meaning of *hasConstValue*(X, C), shown on line 3 of the example, is that the value of X in σ, denoted by $\sigma(X)$, is equal to the value of the constant C, denoted by $\sigma(C)$ (represented internally as $\forall\sigma \in \Sigma_{ed}.\sigma(X) = \sigma(C)$. 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, $\sigma(X)$ will equal $\sigma(C)$. Meanings are used only for reasoning about analyses; they are not used during execution of a dataflow analysis. Henceforth, when we refer to an edge fact schema, we refer to the parameterized fact *and* its meaning.

Propagation rules in Rhodium indicate how edge facts are generated or propagated across CFG nodes. Rules have the form shown in Figure 2. ψ is the antecedent, and f is called the consequent. Rhodium rules must define a static dataflow analysis, so ψ is limited to edge predicates, equality of terms and logical operators.

Edge predicates and terms may be written over IL variables and constants or over metavariables. A metavariable is a variable that can be instantiated to any IL variable, constant, or expression at compile time. Line 1 of Figure 3 declares two metavariables, X and C. X is a metavariable that can be instantiated to any program variable and C is a metavariable that can be instantiated to any constant[1].

Edge predicates are edge fact schema names applied to a sequence of metavariables and IL variables paired with an edge name. The edge name follows the @ sign and indicates the CFG edge on which the edge predicate appears. For example, *hasConstValue*(X, C)@*in* is true if the incoming CFG edge of the current node is annotated with *hasConstValue*(X, C) and *hasConstValue*(X, C)@*out* is true if the outgoing edge of

[1]The convention used throughout this paper is that metavariables C and K range over constants, and W, X, Y, and Z range over IL variables.

a node is annotated with $\mathit{hasConstValue}(X, C)$. Nodes with multiple incoming or outgoing edges have facts annotated with edge numbers as well as edges. For example, branch nodes have two outgoing edges. A fact on the true out edge would be labeled with a $\mathit{out}[0]$, e.g., $\mathit{hasConstValue}(X, C)@\mathit{out}[0]$. Similarly, a fact on the false out edge would be labeled with $\mathit{out}[1]$. *in* and *out* annotations without edge numbers implicitly refer to $\mathit{in}[0]$ and $\mathit{out}[0]$ respectively.

Informally speaking, rules can generate facts or propagate facts. A rule generates a fact if it annotates an out edge with a fact that may not have been on an in edge. The rule on lines 4-5 of Figure 3 generates a $\mathit{hasConstValue}(X, C)$ fact when the current CFG node is an assignment of C to X; $\hat{=}$ is used to indicate that the current statement must be syntactically the same as $X := C$. After X is assigned C, X and C must be equal, and $\mathit{hasConstValue}(X, C)$ can be propagated.

A rule propagates a fact if it annotates an out edge with a fact that was present on an in edge. The rule on lines 6-7 of the example propagates a *hasConstValue* fact across an assignment node: if the fact $\mathit{hasConstValue}(X, C)$ appears on the incoming CFG edge of an assignment node and a variable other than X is assigned to, then the dataflow fact $\mathit{hasConstValue}(X, C)$ should also appear on the outgoing edge of the assignment node.

The semantics of a propagation rule on a CFG node is as follows: for each substitution of the rule's free metavariables that make the antecedent valid at some node in the CFG, the fact in the consequent is propagated. Each propagation rule separately instantiates its free metavariables. For the rule described above, the $\mathit{hasConstValue}(X, C)$ fact will be propagated on the outgoing edge of a node for each substitution of X and C with variables and constants that makes the antecedent valid. The global solution over the entire CFG is the greatest fixed point of the local solutions computed by each propagation rule.

A propagation rule is said to be sound if whenever a fact is propagated on an edge, the meaning of that fact is guaranteed to hold on that edge at runtime. Rhodium dataflow analyses are checked for soundness automatically by discharging a soundness obligation for each propagation rule. For each rule, we ask an automatic theorem prover to show that if the meaning of the antecedent holds before a node for an arbitrary σ, then the meaning of the consequent holds after executing the node in σ. Previous work showed by hand that if all the propagation rules in a Rhodium program pass this condition, then the induced dataflow analysis is sound [12].

Code transformation rules can also be written in Rhodium. A code transformation rule replaces a single CFG node with a different CFG node. The Rhodium verifier used to verify propagation rules can also verify code transformation rules. We do not use code transformation rules in the inference process; the reader interested in more details is referred to the Rhodium technical report [12].

3 Overview of our Approach

The goal of this work is to automatically infer the greatest set of sound dataflow analysis propagation rules given Rhodium fact schemas. The algorithm we have developed infers the rules on lines 4-7 of Figure 3 (and many more) given only the edge fact schema declaration on lines 2-3.

The algorithm starts with a set of edge fact schemas provided by the optimization writer. Each edge fact schema represents a different dataflow analysis that we want to infer. A dataflow analysis for an edge fact schema f with parameters $P_1, \ldots, P_n$ is defined in Rhodium as a set of rules of the form "***if*** ψ ***then*** $f(P_1, \ldots, P_n)@\mathit{out}$". The condition ψ defines some condition on the CFG node and the edges coming into that node that justify annotating the outgoing edge of the node with some instantiation of f.

Without loss of generality, we can assume ψ has the form $\mathit{currStmt} \hat{=} [s] \wedge \phi$ for some CFG statement s and formula ϕ. Rhodium has features that allow one rule to define propagation conditions for multiple statements, but this is no more expressive than having several rules each of which define propagation conditions for one statement. To make sure that the rule can be used in a static analysis, the form of ϕ is subject to the restrictions on the antecedent described in section 2. To find a set of Rhodium rules that propagate some edge fact schema $f(P_1, \ldots, P_n)$, our algorithm tries to find a condition ϕ for each statement such that the rule "***if*** $\mathit{currStmt} \hat{=} [s] \wedge \phi$ ***then*** $f(P_1, \ldots, P_n)@\mathit{out}$" is sound. The P_i metavariables and the metavariables for variables and constants in the statement are all distinct.

We will demonstrate how we find ϕ on an example. Consider the edge fact schema $\mathit{hasConstValue}(X{:}\mathit{Var}, C{:}\mathit{Const})$. We want to find when the meaning $\forall\sigma_{out} \in \Sigma_{out}.\sigma_{out}(X) = \sigma_{out}(C)$ will be true on the edge out of a node $[Z := K]$. ϕ should be a precondition[2] of $\forall\sigma_{out} \in \Sigma_{out}.\sigma_{out}(X) = \sigma_{out}(C)$ with respect to the statement $[Z := K]$. To find the most general rule, our algorithm starts by finding the weakest precondition[3].

Ignoring the possibility of pointers for the moment, the weakest precondition of a predicate P with respect to a statement $[Z := K]$ can be found by replacing all instances of Z in the predicate with K; this says that if some predicate was true of K before $[Z := K]$ was executed, it will be true of Z afterward. At first glance, it may seem that this leaves nothing to be done for the examples predicate $\forall\sigma_{out} \in \Sigma_{out}.\sigma_{in}(X) = \sigma_{in}(C)$, since $\forall\sigma_{out} \in \Sigma_{out}.\sigma_{in}(X) = \sigma_{in}(C)$ does not mention Z. However, X and Z are metavariables that may be instantiated to the same program variable at compile time. To account for this, the weakest precondition must have two cases; one for when X and Z are instantiated to the same program variable and one for when they are not. In the former situation, all references to X in the predicate should be replaced by K, while in the later situation the predicate should be left unchanged. Recall that $E \doteq E'$ tests E and E' for syntactic equality. The weakest precondition for this example is therefore

$$\forall\sigma_{in} \in \Sigma_{in}.((X \doteq Z \wedge \sigma_{in}(K) = \sigma_{in}(C)) \vee (X \not\doteq Z \wedge \sigma_{in}(X) = \sigma_{in}(C))$$

That condition says that if X and Z are instantiated to the same variable, then the constants K and C must have the same value before the assignment. If X and Z are not instantiated to the same program variable, then X and C must have the same value before the assignment. The weakest precondition must hold over program states valid on the edge coming into the assignment node.

The weakest precondition above cannot be used directly as the condition ϕ in a rule, because it does not satisfy the restrictions imposed on Rhodium propagation rules to ensure they can be evaluated at compile time. ϕ can contain only syntactic equality and edge predicates. It cannot contain explicit references to the runtime state σ_{in}. Our algorithm converts the weakest precondition to a valid ϕ using rewrites. A rewrite is a rule that transforms one predicate into another, possibly stronger[4] predicate. Rewriting terminates when the resulting predicate, ϕ, satisfies the restrictions of a Rhodium propagation rule. Because rewriting always leads to a predicate that is equivalent to or stronger than the initial predicate, the resulting formula ϕ will imply the weakest precondition, ensuring the inferred rule is sound.

For the formula above, two rewrites are used. First, two constants have the same value if and only if they are syntactically the same constant, so $\sigma_{in}(K) = \sigma_{in}(C)$ can be rewritten to $K \doteq C$. Doing that and dropping the $\forall\sigma_{in} \in \Sigma_{in}$ where it is unnecessary gives

$$(X \doteq Z \wedge K \doteq C) \vee (X \not\doteq Z \wedge \forall\sigma_{in} \in \Sigma_{in}.\sigma_{in}(X) = \sigma_{in}(C))$$

$\sigma_{in}(X) = \sigma_{in}(C)$ cannot be simplified to syntactic comparison. However, $\forall\sigma_{in} \in \Sigma_{in}.\sigma_{in}(X) = \sigma_{in}(C)$ matches the meaning of the *hasConstValue* edge fact schema. The meaning of a fact can be rewritten to a corresponding edge predicate, so $\forall\sigma_{in} \in \Sigma_{in}.\sigma_{in}(X) = \sigma_{in}(C)$ can be rewritten to $\mathit{hasConstValue}(X, C)@\mathit{in}$. Applying this rewrite gives

$$(X \doteq Z \wedge K \doteq C) \vee (X \not\doteq Z \wedge \mathit{hasConstValue}(X, C)@\mathit{in})$$

This formula uses only syntactic comparison and edge predicates, so it is a valid ϕ for our Rhodium rule. The condition above gives the Rhodium rule

```
if  $\mathit{currStmt} \doteq [Z := K] \wedge$
        $((X \doteq Z \wedge K \doteq C) \vee (X \not\doteq Z \wedge \mathit{hasConstValue}(X, C)@\mathit{in}))$
then  $\mathit{hasConstValue}(X, C)@\mathit{out}$
```

The rule above can be split into two rules

[2]A formula P is a precondition of a formula Q with respect to some statement s if whenever P is true before the execution of s, Q must be true afterward. Q is called the postcondition.

[3]A precondition P is the weakest precondition if for all preconditions P', $P' \Rightarrow P$.

[4]A predicate P is stronger than a predicate Q if P implies Q.

```
    function $GenerateRules(decls\text{: } set[FactSchemaDecl])\text{: } set[Rule]$
1.      let $results := \emptyset$
2.      for each "$\textbf{\textit{defined edge fact}}\ F\ \textbf{\textit{with meaning}}\ M$" $\in decls$ do
3.          for each statement form $S$ do
4.              for each edge $out \in succ(S)$ do
5.                  let $\phi := wp(S, out, M)$
6.                  let $\psi := RemoveRuntimeState(\phi, decls)$
7.                  if $\psi \neq false$ then
8.                      let $rule :=$ "$\textbf{\textit{if}}\ currStmt = [s] \wedge \psi\ \textbf{\textit{then}}\ F@out$"
9.                      $results := results \cup \{rule\}$
10.     return $results$

    function $RemoveRuntimeState(\phi\text{: } Formula,$
                                        $decls\text{: } set[FactSchemaDecl])\text{: } Formula$
11.     let $\phi_{simp} := Simplify(\phi)$
12.     if $\phi_{simp}$ contains no $\sigma_{in}$ then
13.         return $\phi_{simp}$
14.     if runs too long then
15.         return $false$
16.     let $\phi_{opt} := Strengthen(\phi_{simp}, decls)$
17.     return $RemoveRuntimeState(\phi_{opt}, decls)$
```

Figure 4: Algorithm for generating rules from fact declarations

$\textbf{\textit{if}}\ \ currStmt \doteq [Z := K] \wedge (X \doteq Z \wedge K \doteq C)$
$\textbf{\textit{then}}\ \ hasConstValue(X, C)@out$

$\textbf{\textit{if}}\ \ currStmt \doteq [Z := K] \wedge (X \not\doteq Z \wedge hasConstValue(X, C)@in)$
$\textbf{\textit{then}}\ \ hasConstValue(X, C)@out$

The condition $currStmt \doteq [Z := K] \wedge (X \doteq Z \wedge K \doteq C)$ is equivalent to $currStmt \doteq [X := C]$. Thus, the inferred rules above are equivalent to the rules on lines 4-5 and 6-7 of Figure 3.

In this example, σ_{in} was quickly removed from the weakest precondition, but it may not always be this simple. Removing σ_{in} from the weakest precondition may require strengthening the formula, i.e., rewriting an intermediate formula ϕ to ϕ' where ϕ' implies ϕ but ϕ does not imply ϕ'. It may also involve rewrites that do not immediately remove σ_{in} but make it easier to remove some reference to σ_{in} later. The next section gives details on how our algorithm finds the weakest precondition and how it removes σ_{in} to get valid Rhodium rules.

4 Algorithm for Inferring Rules from Facts

Figure 4 shows pseudo-code for the Rhodium rule inference algorithm. The function *GenerateRules* (lines 1-10) takes a set of fact schema declarations and returns a set of propagation rules. Each fact declaration has the form "***define edge fact*** F ***with meaning*** M". An edge fact schema, F, consists of a fact name and a typed list of Rhodium variable arguments to the fact. The meaning M is a first-order logic formula.

For each fact definition, the algorithm iterates through each IL statement form s and out edge *out* for that statement form and finds rules that propagate the fact F on edge *out* of statement s. This is done by first finding the weakest precondition of the fact meaning M with respect to s and *out*. The weakest precondition may reference the program state σ_{in}. References to σ_{in} cannot appear in propagation rules,

$$
\begin{array}{lcl}
 & (Y \overset{\circ}{=} X \wedge K \overset{\circ}{=} C) \;\vee (Y \overset{\circ}{\neq} X \wedge \mathit{hasConstValue}(X,C)@\mathit{in}) & \\
\hline
(5) & (\forall\sigma_{in}.\big[Y \overset{\circ}{=} X\big] \wedge \forall\sigma_{in}.\big[K \overset{\circ}{=} C\big]) \;\vee (\forall\sigma_{in}.\big[Y \overset{\circ}{\neq} X\big] \wedge \forall\sigma_{in}.\big[\sigma_{in}(X) = \sigma_{in}(C)\big]) & \mathit{fact\ match}, \mathit{logSimp} \\
\hline
(4) & \forall\sigma_{in}.\big[Y \overset{\circ}{=} X \wedge K \overset{\circ}{=} C\big] \;\vee \forall\sigma_{in}.\big[Y \overset{\circ}{\neq} X \wedge \sigma_{in}(X) = \sigma_{in}(C)\big] & \mathit{logSimp} \\
\hline
(3) & \forall\sigma_{in}.\big[(Y \overset{\circ}{=} X \wedge K \overset{\circ}{=} C) \;\vee (Y \overset{\circ}{\neq} X \wedge \sigma_{in}(X) = \sigma_{in}(C))\big] & \mathit{case} \\
\hline
(2) & \forall\sigma_{in}.\big[(\sigma_{in}(\&Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(K) = \sigma_{in}(C)) \;\vee (\sigma_{in}(\&Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big] & \overset{\circ}{=}_{\&}, \overset{\circ}{\neq}_{\&}, \overset{\circ}{=}_{c} \\
\hline
(1) & \forall\sigma_{out} \in \Sigma_{out}.\, (\sigma_{out}(X) = \sigma_{out}(C)) & wp(Y := K)
\end{array}
$$

Figure 5: Inference steps for $\mathit{hasConstValue}(X, C)$ for statements of the form $Y := K$

$$
\begin{array}{lcl}
 & (\mathit{mustPointTo}(Y,X)@\mathit{in} \wedge \mathit{hasConstValue}(Z,C)@\mathit{in}) \vee & \\
 & (\mathit{mustNotPointTo}(Y,X)@\mathit{in} \wedge \mathit{hasConstValue}(X,C)@\mathit{in}) \vee & \\
 & (\mathit{hasConstValue}(Z,C)@\mathit{in} \wedge \mathit{hasConstValue}(X,C)@\mathit{in}) & \\
\hline
(5) & (\forall\sigma_{in}.\big[\sigma_{in}(Y) = \sigma_{in}(\&X)\big] \wedge \forall\sigma_{in}.\big[\sigma_{in}(Z) = \sigma_{in}(C)\big]) \vee & \mathit{fact\ match} \\
 & (\forall\sigma_{in}.\big[\sigma_{in}(Y) \neq \sigma_{in}(\&X)\big] \wedge \forall\sigma_{in}.\big[\sigma_{in}(X) = \sigma_{in}(C)\big]) \vee & \\
 & (\forall\sigma_{in}.\big[\sigma_{in}(Z) = \sigma_{in}(C)\big] \wedge \forall\sigma_{in}.\big[\sigma_{in}(X) = \sigma_{in}(C)\big]) & \\
\hline
(4) & \forall\sigma_{in}.\big[\sigma_{in}(Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(Z) = \sigma_{in}(C)\big] \vee & \mathit{logSimp} \\
 & \forall\sigma_{in}.\big[\sigma_{in}(Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C)\big] \vee & \\
 & \forall\sigma_{in}.\big[\sigma_{in}(Z) = \sigma_{in}(C) \wedge \sigma_{in}(X) = \sigma_{in}(C)\big] & \\
\hline
(3) & \forall\sigma_{in}.\big[(\sigma_{in}(Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(Z) = \sigma_{in}(C)) \;\vee (\sigma_{in}(Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big] & \forall\ \mathit{resolution} \\
\hline
(1) & \forall\sigma_{out} \in \Sigma_{out}.\, (\sigma_{out}(X) = \sigma_{out}(C)) & wp(*Y := Z)
\end{array}
$$

Figure 6: Inference steps for $\mathit{hasConstValue}(X, C)$ for statements of the form $*Y := Z$

so the *RemoveRuntimeState* function is used to do rewrites that remove references to the program state. If it requires too many calls to *RemoveRuntimeState* to remove all reference to σ, the function terminates by returning *false*. If all references to the program state are successfully removed without reducing the precondition to just $false$, a new rule is created.

The function *RemoveRuntimeState* (lines 11-15) removes references to σ_{in} by applying rewrites. A rewrite replaces replaces a formula ϕ with a formula ϕ'. A rewrite from phi to phi' can be seen as a single step in a backwards inference system. That is, rewriting ϕ to ϕ' can be seen as the inference step

$$\frac{\phi'}{\phi}$$

The sequence of rewrites is a proof that the condition found by the algorithm implies the weakest precondition; this implies that the rules found by the algorithm are sound.

The algorithm uses two types of rewrites. Simplification rewrites transform a formula to a semantically equivalent but syntactically simpler formula. When simplification rewrites cannot remove σ_{in}, the algorithm calls the applies strengthening rewrites which transform a formula to a stronger, i.e., less precise, formula that is closer to satisfying the restrictions on ϕ. The algorithm tries all strengthening rewrites on a formula and returns a disjunction of the rewritten formulas. If no strengthening rewrites apply, the result is *false*. This process is repeated until the formula no longer directly references the run-time program state. The algorithm tries all possible strengthening rewrites because each strengthening rewrite loses precision in a different way; returning all options retains more precision than choosing one option.

The algorithm uses the functions *wp*, *Simplify*, and *Strengthen*. *wp* finds the weakest precondition of a formula with respect to a particular statement. *Simplify* performs rewrites that transform a formula to a syntactically simpler but semantically equivalent formula. *Strengthen* takes a formula and returns a stronger formula which approximates the original formula.

The examples in Figures 5 and 6 show how to derive rules for the fact declaration "$hasConstValue(X, C)$ **with meaning** $\sigma_{in}(X) = \sigma_{in}(C)$" for the statements $[Y := K]$ and $[*Y := Z]$. The details are explained in the following sections.

The examples are shown as proof trees. The bottom line shows meaning of $hasConstValue(X, C)$; the top line shows the final result. Each line represents one or more steps taken in one of the helper functions described above. The label on the left numbers the steps in the inference; the label on the right indicates exactly what was done in each step. The examples use the shorthand $\forall\sigma_{in}.\big[P\big]$ to stand for $\forall\sigma_{in} \in \Sigma_{in}.P$. Σ_{in} is the set of program states that can occur at runtime on an edge into a CFG node. We also define Σ_{out} as the set of program states that can occur at runtime on an edge out of a CFG node.

4.1 Computing the weakest precondition

Let $wp(s, out, \alpha)$ denote the *weakest liberal precondition* of a predicate α with respect to the outgoing edge *out* of a statement s. $wp(s, out, \alpha)$ is the weakest predicate[5] such that if $wp(s, out, \alpha)$ is true before the execution of s, then α will be true on edge *out* after s terminates, assuming s terminates and reaches edge *out*.

The traditional method for calculating $wp(s, out, \alpha)$ directly manipulates the expressions in α based on the effect of s [8]. However, direct manipulation becomes increasingly complicated in the presence of aliasing, e.g., via pointers. The approach take by Ball *et al.* [2] tries extending direct manipulation to a domain with pointers, but we have found that their approach does not always give the correct results, e.g., for pointers that can point to themselves.. Complex statements such as declarations, heap allocation and arrays also further complicate the direct manipulation approach.

To fix this, we compute the weakest precondition using an approach based on work by Cartwright and Oppen [3]. Instead of thinking of the weakest precondition calculation as a manipulation of the syntactic form of a predicate, the weakest precondition is computed by finding changes to the environment and memory caused by the statement. This is possible because σ_{out} and σ_{in} are not arbitrary program states; they are

[5]A predicate Q is said to be weaker than a predicate P if $P \Rightarrow Q$.

```
    function $wp(s\text{: } Stmt, out\text{:}edge, \alpha_{out}\text{:}Pred_{\sigma_{in}})\text{:}Pred_{\sigma_{out}}$
1.      let $\alpha_{out_{EM}} : Pred := ConvertToMemEnv(\alpha_{out})$
2.      let $\alpha_{in_{EM}} : Pred := OutStateToInState(\alpha_{out_{EM}}, s)$
3.      $\alpha_{simp} : Pred := SimplifyLookups(\alpha_{in_{EM}})$
4.      let $\alpha_{in} : Pred := ConvertToSigma(\alpha_{in_{EM}})$
5.      let $\alpha_{ns} : Pred := AddNotStuck(s, out, \alpha_{in})$
6.      return $\alpha_{ns}$
```

Figure 7: Algorithm for finding the weakest precondition.

related by the forward semantics of the statement. The forward semantics of all the statements in our IL can be found in the technical report for the Rhodium language [12].

The procedure for finding the weakest precondition is found in Figure 7. The procedure takes a statement s, an out edge *out*, and a predicate, α_{out}, in terms of σ_{out}. The procedure has four steps. In the first step, $ConvertToMemEnv(\alpha_{out})$, references to σ_{out} are expanded into a references to an explicit program environment E_{out} and program memory M_{out}. In the second step, $OutStateToInState(\alpha_{out_{EM}}, s)$, the outgoing environment and memory are reexpressed as functions of the incoming environment E_{in} and memory M_{in}. The third step, $SimplifyLookups(\alpha_{in_{EM}})$, simplifies selections from updated environments and memories and from new arrays. After simplifying the predicate, $ConvertToSigma(\alpha_{simp})$ merges the memory and environment back into one program state σ_{in}. Finally, $AddNotStuck(s, out, \alpha_{in})$ adds assumptions that the statement is not stuck. These steps are explained in detail in the next sections.

4.1.1 $ConvertToMemEnv(\alpha_{out})$

A program state σ encapsulates both the program environment and the program memory. Let E represent a program environment at a particular program point. E is a mapping from variable names to locations (i.e., addresses). Looking up variable x in environment E is denoted $E[x]$. Let M represent a program memory at a particular program point. M is a mapping from locations to values. Looking up a location ℓ is a memory M is denoted $M[\ell]$. Finally, if a is a pointer to an array and i is an index in bounds of the array then $a[i]$ is the address of the ith element of a. Accessing an array out of bounds is a fatal error that will prevent execution from reaching the successor edge being computed.

In the notation we have been using, the address of a variable has been denoted $\sigma(\&X)$ and the value of a variable X has been denoted $\sigma(X)$. When reexpressed as functions of M and E they are denoted $E[X]$ and $M[E[X]]$ respectively; to find the value of X, look up X in the environment to get its address, and then look up that address in memory to get the corresponding value. Figure 8 shows how to translate between σ-notation and E/M-notation for all expressions in the IL. $[\![\mathcal{E}]\!]_R(E, M)$ translates a right-hand-side expression to one expressed as a function of M and E. $[\![\mathcal{E}]\!]_L(E, M)$ does the same for left-hand-side expressions. Finally, we use these translations to convert a predicate to one using more explicit E/M-notation by replacing all occurrences of $\sigma_{out}(\mathcal{E})$ with $[\![\mathcal{E}]\!]_R(E_{out}, M_{out})$. Any quantifiers over σ_{out} must be replaced with a quantifier over E_{out} and M_{out}.

Example. In Figure 5 the first step in finding $wp([Y := K], 0, \forall\sigma_{out} \in \Sigma_{out}.\sigma_{out}(X) = \sigma_{out}(C))$ is to convert $\sigma_{out}(X) = \sigma_{out}(C)$ to $\forall E_{out}, M_{out}.[\![X]\!]_R(E_{out}, M_{out}) = [\![C]\!]_R(E_{out}, M_{out})$. This becomes $\forall E_{out}, M_{out}.M_{out}[E_{out}[X]] = C$. This conversion does not depend on the statement and is the same for the example in Figure 6 when finding $wp([*Y := Z], 0, \forall\sigma_{out} \in \Sigma_{out}.\sigma_{out}(X) = \sigma_{out}(C))$.

4.1.2 $OutStateToInState(\alpha_{out_{EM}}, s, out)$

α in terms of M_{out} and E_{out} is exactly the condition that needs to be true after statement s executes. To determine the weakest precondition, α must be re-expressed in terms of E_{in} and M_{in} where E_{in} and M_{in}

$$\sigma(\mathcal{E}) \rightarrow [\![\mathcal{E}]\!]_R(E,M)$$

$$\begin{array}{lcl}
[\![C]\!]_R(E,M) & = & C \\
[\![\&X]\!]_R(E,M) & = & E[X] \\
[\![X]\!]_R(E,M) & = & M[[\![\&X]\!]_R(E,M)] \\
[\![*X]\!]_R(E,M) & = & M[[\![X]\!]_R(E,M)] \\
[\![A \ op \ B]\!]_R(E,M) & = & [\![A]\!]_R(E,M) \ op \ [\![B]\!]_R(E,M) \\
[\![A[I]]\!]_R(E,M) & = & [\![A]\!]_R(E,M)[[\![I]\!]_R(E,M)] \\
 & & \\
[\![X]\!]_L(E,M) & = & [\![\&X]\!]_R(E,M) \\
[\![*X]\!]_L(E,M) & = & [\![X]\!]_R(E,M)
\end{array}$$

Figure 8: Translation from σ-notation to E/M-notation.

are the environment and memory valid on the edge into the statement s. The relationship between E_{in}/M_{in} and E_{out}/M_{out} is not arbitrary; they are related by the forward semantics of s in the IL. Figure 9 shows the mapping between E_{in}/M_{in} and E_{out}/M_{out} for all different statement types. These relationships capture the effect of a statement on the environment and memory. The behavior of a statement depends on neither the *out* edge nor the predicate under consideration.

The relationship uses two update functions, one for updating an environment and another for updating a memory. $upd_E(E, X, \ell)$ takes an environment E, a variable name X, and a location ℓ and returns a new environment that maps X to ℓ and all other names to the locations they are mapped to in E. $upd_M(M, \ell, v)$ takes a memory M, a location ℓ, and a value v and returns a new memory that maps ℓ to v and all other locations to the values they are mapped to in M. Either update function may take a list of names/locations and locations/values to indicate that all the elements of the first list are simultaneously mapped to the elements of the second. *newloc* and $newloc_i$ represent fresh locations that do not exist in M or E. The mappings also use the function $newarray(\mathcal{E}, len)$. This function takes an expression which evaluates to the location of the array in memory and an integer *len* and returns a new array value, which maps integers in the range $[0 \ldots len)$ to fresh locations $newloc_i$. $\mathcal{E}$ is only used when translating back to σ-notation because σ-notation cannot use *newarray* . Accessing a uninitialized piece of memory is a fatal error that will prevent execution from reaching the successor edge being computed. *error* is a placeholder for any value which results in a fatal error when read.

As another example, consider the relationship between the memory and environment coming into an assignment $X := \mathcal{E}$ and those going out of that assignment. The statement does not modify or create any addresses, so $E_{out} = E_{in}$. The statement does modify the memory. The memory associated with $\&X$ now contains the value obtained by evaluating $\mathcal{E}$ in E_{in} and M_{in}; all other memory locations contain the same value as in the incoming memory. This is denoted $M_{out} = upd_M(M_{in}, [\![X]\!]_L(E_{in}, M_{in}), [\![\mathcal{E}]\!]_R[E_{in}, M_{in}])$; the outgoing memory is the ingoing memory that has been updated so that the location indicated by x contains the value indicated by $\mathcal{E}$.

Consider the changes to the memory and environment caused by an array declaration, *decl* $X[I]$. An array declaration is the most complicated statement in the IL. The statement adds a new entry to E_{in}; X is now mapped to some fresh location represented by *newloc*. In the memory, *newloc* is mapped to a new array of length $len = [\![I]\!]_R(E_{in}, M_{in})$. Selecting from the array at index i gives the location at index i if $0 \leq i < I$ and is undefined otherwise. The value stored at any $newloc_i$ is undefined. If $s = [decl\ X[I]]$ and $out = 0$ then this is represented as $E_{out} = upd_E(E_{in}, X, newloc)$ and $M_{out} = upd_M(M_{in}, [newloc, s], [newarray(\&X, len), error, \ldots, error])$

Figure 9 shows the relationship between E_{in} and E_{out} and M_{in} and M_{out} for all statement types. These relationships are nearly the same as the specification of the forward semantics for these statements in the language. The only difference is that fresh locations are indicated symbolically as *newloc* and $newloc_i$. This representation of freshly allocated locations works because we find only the weakest precondition over a

$s = [skip]$:	$E_{out} = E_{in}$
	$M_{out} = M_{in}$
$s = [decl\ X]$:	$E_{out} = upd_E(E_{in}, X, newloc)$
	$M_{out} = upd_M(M_{in}, newloc, error)$
$s = [decl\ X[I]]$:	$E_{out} = upd_E(E_{in}, X, newloc)$
	$M_{out} = upd_M(M_{in}, [newloc, newloc_0, \dots, newloc_{len-1}],$ $[newarray(\&X, len), error, \dots, error])$
$s = [X := new]$:	$E_{out} = E_{in}$
	$M_{out} = upd_M(upd_M(M_{in}, [\![X]\!]_L(E_{in}, M_{in}), newloc), newloc, uninit)$
$s = [X := new[I]]$:	$E_{out} = E_{in}$
	$M_{out} = upd_M(M_{in}, [[\![X]\!]_L(E_{in}, M_{in}), newloc, newloc_0, \dots, newloc_{len-1}],$ $[newloc, newarray(X, len), error, \dots, error])$
$s = [L := \mathcal{E}]$:	$E_{out} = E_{in}$
	$M_{out} = upd_M(M, [\![L]\!]_L(E_{in}, M_{in}), [\![\mathcal{E}]\!]_R(E_{in}, M_{in}))$
$s = [if\ B \dots]$:	$E_{out} = E_{in}$
	$M_{out} = M_{in}$

Figure 9: Environment and Memory updates

$$\begin{array}{lcl} upd_E(E, X, \ell)[Y] & = & \text{if } X \doteq Y \text{ then } \ell \text{ else } E[Y] \\ upd_M(M, \ell_1, v)[\ell_2] & = & \text{if } \ell_1 = \ell_2 \text{ then } v \text{ else } M[\ell_2] \\ newarray(\mathcal{E}, len)[i] & = & \text{if } 0 \leq i \wedge i < len \text{ then } \ell_i \text{ else } error \end{array}$$

Figure 10: Simplifying lookups

single statement at a time. To extend this approach to handle sequencing of statements, the relationship between E_{in}/M_{in} and E_{out}/M_{out} would have to make sure that the syntactic representation of all locations allocated by the sequence of statements are distinct.

Example. In Figure 5 the algorithm is finding the weakest precondition for $[Y := K]$, an assignment of a constant to a variable. For this statement, $E_{out} = E_{in}$ and $M_{out} = upd_M(M_{in}, E_{in}[Y], K)$. That is, the statement does not change the environment (no new variables are declared) and changes the memory only by updating the address of Y to hold the value K. The translated meaning of *hasConstValue*, $\forall E_{out}, M_{out}.M_{out}[E_{out}[X]] = C$, becomes $\forall E_{in}, M_{in}.upd_M(M_{in}, E_{in}[Y], K)[E_{in}[X]] = C$.

Example. The example from Figure 6, finds the weakest precondition over the statement $[*Y := Z]$. For this statement $E_{out} = E_{in}$ and $M_{out} = upd_M(M_{in}, M_{in}[E_{in}[Y]], M_{in}[E_{in}[Z]])$. That is, the statement updates the location Y points to, whose address is stored in Y, to hold the value stored in Z. Thus $\forall E_{out}, M_{out}.M_{out}[E_{out}[X]] = C$ becomes $\forall M_{in}, E_{in}.upd_M(M_{in}, M_{in}[E_{in}[Y]], M_{in}[E_{in}[Z]])[E_{in}[X]] = C$.

4.1.3 *SimplifyLookups*$(\alpha_{in_{EM}})$

The next step in the algorithm is to replace lookups from updated environments, memories, and arrays with values or lookups directly from E_{in} and M_{in}. A selection from an updated environment, $upd_E(E, X, \ell)[Y]$, is simplified to if $X \doteq Y$ then ℓ else $E[Y]$. Lookups from updated memories are similar except that we case on whether the location selected is the updated location. Indexing a new array is reduced to a similar form but the the cases are on whether the index is within the bounds of the array. Lookup simplification rules are shown in Figure 10.

The simplified rules can introduce conditionals inside of terms. The final step in the simplification hoists the conditionals outside the terms using the rewrite $T[\text{if } P \text{ then } \mathcal{E}_1 \text{ else } \mathcal{E}_2]$ to $(P \wedge T[\mathcal{E}_1]) \vee (\neg P \wedge T[\mathcal{E}_2])$, where $T[\mathcal{E}]$ is the largest term enclosing $\mathcal{E}$.

$$\mathcal{T}_{E/M} \rightarrow \sigma(\llbracket \mathcal{T}_{E/M} \rrbracket_\sigma)$$

$$\begin{array}{lcl}
\llbracket C \rrbracket_\sigma & = & C \\
\llbracket X \rrbracket_\sigma & = & X \\
\llbracket \mathit{error} \rrbracket_\sigma & = & \mathit{error} \\
\llbracket \mathit{newloc} \rrbracket_\sigma & = & \mathit{newloc} \\
\llbracket E[X] \rrbracket_\sigma & = & \&X \\
\llbracket M[\mathcal{E}], \sigma \rrbracket_\sigma & = & *\llbracket \mathcal{E} \rrbracket_\sigma \\
\llbracket \mathit{newarray}(\mathcal{E}, s) \rrbracket_\sigma & = & *\llbracket \mathcal{E} \rrbracket_\sigma \\
\llbracket \mathcal{E}_1 \ \mathit{op} \ \mathcal{E}_2 \rrbracket_\sigma & = & \llbracket \mathcal{E}_1 \rrbracket_\sigma \ \mathit{op} \ \llbracket \mathcal{E}_2 \rrbracket_\sigma \\
\llbracket \mathcal{E}_1[\mathcal{E}_2] \rrbracket_\sigma & = & \llbracket \mathcal{E}_1 \rrbracket_\sigma[\llbracket \mathcal{E}_2 \rrbracket_\sigma]
\end{array}$$

Figure 11: Translation from E/M-notation to σ-notation.

Example. For the example from Figure 5, $\mathit{upd}_M(M_{in}, E_{in}[Y], K)[X] = C$ simplifies to

$$\forall E_{in}, M_{in}.(\text{if } E_{in}[Y] = E_{in}[X] \text{ then } K \text{ else } M_{in}[E_{in}[X]]) = C.$$

Hoisting out the nested conditional yields

$$\forall E_{in}, M_{in}.((E_{in}[Y] = E_{in}[X] \wedge K = C) \vee (E_{in}[Y] \neq E_{in}[X] \wedge M_{in}[E_{in}[X]] = C))$$

This says that if X and Y refer to the same location, then K and C must be the same constant. If they have different values in the environment, then the value of X in the incoming environment and memory must be the same as the value of C.

Example. In the example in Figure 6, $\mathit{upd}_M(M_{in}, M_{in}[E_{in}[Y]], M_{in}[E_{in}[Z]])[X] = C$ simplifies to

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

Hoisting out the nested conditional yields

$$\forall E_{in}, M_{in}.((M_{in}[E_{in}[Y]] = E_{in}[X] \wedge M_{in}[E_{in}[Z]] = C) \vee (M_{in}[E_{in}[Y]] \neq E_{in}[X] \wedge M_{in}[E_{in}[X]] = C))$$

This condition says that if the value held by Y is the location of X then the value of Z in the incoming environment and memory must be equal to C. If they are not equal then the value of X must equal C.

4.1.4 *ConvertToSigma*(α_{simp})

The final step is to convert the predicate from E/M-notation to σ-notation. Each E/M-notation term $\mathcal{T}$ is in the formula is replaced with $\sigma_{in}(\llbracket \mathcal{T} \rrbracket_\sigma)$. $\llbracket T \rrbracket_\sigma$ is shown in Figure 11. Once terms have been converted back to σ-notation, quantifiers over E_{in} and M_{in} must be covered to quantifiers over σ_{in}.

Example. For the example from Figure 5,

$$\forall E_{in}, M_{in}.((E_{in}[Y] = E_{in}[X] \wedge K = C) \vee (E_{in}[Y] \neq E_{in}[X] \wedge M_{in}[E_{in}[X]] = C))$$

translates to

$$\forall \sigma_{in}.\big[\sigma_{in}(\&Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(K) = \sigma_{in}(C)) \vee (\sigma_{in}(\&Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big]$$

Example. For the example from Figure 6,

$$\forall E_{in}, M_{in}.((M_{in}[E_{in}[Y]] = E_{in}[X] \wedge M_{in}[E_{in}[Z]] = C) \vee (M_{in}[E_{in}[Y]] \neq E_{in}[X] \wedge M_{in}[E_{in}[X]] = C))$$

translates to

$$\forall \sigma_{in}.\big[(\sigma_{in}(Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(Z) = \sigma_{in}(C)) \vee (\sigma_{in}(Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big]$$

$$
\begin{array}{lcl}
ns_s(\texttt{skip}, 0) & = & \mathit{true} \\
ns_s(\texttt{decl}\ X, 0) & = & ns_e(X) \\
ns_s(\texttt{decl}\ A[I], 0) & = & ns_e(A) \wedge \mathit{isArray}(A) \\
ns_s(X := \texttt{new}, 0) & = & ns_e(X) \\
ns_s(X := \texttt{new}[I], 0) & = & ns_e(X) \\
ns_s(X := \mathcal{E}, 0) & = & ns_e(X) \wedge ns_e(\mathcal{E}) \\
ns_s(*X := Y, 0) & = & ns_e(*X) \wedge ns_e(Y) \\
ns_s(\texttt{if}\ B\ \texttt{goto}\ \iota_1\ \texttt{else}\ \iota_2, 0) & = & ns_e(B) \wedge \sigma(B) = \sigma(\texttt{true}) \\
ns_s(\texttt{if}\ B\ \texttt{goto}\ \iota_1\ \texttt{else}\ \iota_2, 1) & = & ns_e(B) \wedge \sigma(B) = \sigma(\texttt{false}) \\
\\
ns_e(C) & = & \mathit{true} \\
ns_e(X) & = & ns_e(\&X) \\
ns_e(\&X) & = & ns_e(\&X) \\
ns_e(*X) & = & ns_e(*X) \\
ns_e(A\ \mathit{op}\ B) & = & (\exists C_1 : \mathit{Const}.\sigma(C_1) = \sigma(A)) \wedge (\exists C_2 : \mathit{Const}.\sigma(C_2) = \sigma(B)) \\
ns_e(A[I]) & = & \mathit{isArray}(*A) \wedge (\sigma(\texttt{0}) \leq \sigma(I)) \wedge (\sigma(I) < \mathit{arrayLength}(\sigma(*A))) \\
ns_e(\mathit{newloc}) & = & \mathit{true} \\
ns_e(\mathit{uninit}) & = & \mathit{false}
\end{array}
$$

Figure 12: Definition of ns

4.1.5 $\mathit{AddNotStuck}(s, \mathit{out}, \alpha_{in})$

We compute the weakest liberal precondition under the assumption that the statement terminates and proceeds to the outgoing edge of the node in the CFG. A statement or term that does not terminate or terminates with an error is *stuck*. Non-stuckness assumptions are not usually explicitly represented in the weakest precondition; such predicates are useful only if a predicate explicitly mentions terms that are not stuck. We represent these statements explicitly. $\mathit{AddNotStuck}(s, \mathit{out}, \alpha_{in})$ takes a predicate of the form $\forall \sigma_{in}.\big[P\big]$ and returns $\forall \sigma_{in}.\big[ns_s(s, \mathit{out}) \implies P\big]$. $ns_s(s, \mathit{out})$ is true when the statement s is not stuck on the edge out. Figure 12 defines $ns_s(s, \mathit{out})$ in terms of $ns_e(\mathcal{E})$ (true when the expression $\mathcal{E}$ is not stuck) and $\mathit{isArray}(\mathcal{E})$ (true when $\mathcal{E}$ is an array),

Example. For the example in Figure 5,

$$\forall \sigma_{in}.\big[(\sigma_{in}(\&Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(K) = \sigma_{in}(C)) \vee (\sigma_{in}(\&Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big]$$

becomes the final weakest liberal precondition

$$\forall \sigma_{in}.\big[ns_e(\&Y) \implies (\sigma_{in}(\&Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(K) = \sigma_{in}(C)) \vee (\sigma_{in}(\&Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big]$$

after expanding $ns_s([Y := K])$. This is the precondition shown on top of line (1) in the figure.

Example. For the example in Figure 6,

$$\forall \sigma_{in}.\big[(\sigma_{in}(Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(Z) = \sigma_{in}(C)) \vee (\sigma_{in}(Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big]$$

becomes the final weakest liberal precondition

$$\forall \sigma_{in}.\big[(ns_e(*Y) \wedge ns_e(\&Z)) \implies (\sigma_{in}(Y) = \sigma_{in}(\&X) \wedge \sigma_{in}(Z) = \sigma_{in}(C)) \vee (\sigma_{in}(Y) \neq \sigma_{in}(\&X) \wedge \sigma_{in}(X) = \sigma_{in}(C))\big]$$

after expanding $ns_s([*Y := Z])$. This is the precondition shown on top of line (1) in the figure.

4.2 Rewrite contexts

The helper functions *Simplify* and *Strengthen* define rewrites on formulas or terms that occur in a context. The position at which the rewrite occurs is a hole in the formula. $\mathcal{F}^t[\cdot]$ and $\mathcal{F}[\cdot]$ represent the different contexts in which a rewrite can occur. $\mathcal{F}^t[\cdot]$ represents a term hole in the formula $\mathcal{F}$. $\mathcal{F}[\cdot]$ represents a formula hole in $\mathcal{F}$. If a formula hole must occur in a positive or a negative position of a formula[6], it is denoted as $\mathcal{F}|^+[\cdot]$ or $\mathcal{F}^-[\cdot]$ respectively.

4.3 Simplifying

The function *Simplify* takes a formula and returns a semantically equivalent but syntactically more regular formula. Simplification is expressed with simplification rewrite rules. Some of the simplification rewrite rules are shown in Figure 13. A simplification rewrite has the form $\mathcal{T}_1 \rightsquigarrow_T \mathcal{T}_2$ for terms and $\mathcal{F}_1 \rightsquigarrow \mathcal{F}_2$ for formulas and rewrites a formula to a semantically equivalent formula. Simplification rules are applied until no more rewrites are applicable.

The *Simplify* function performs two types of simplification rewrites: IL simplification rewrites and logical simplification rewrites. We verified that all rewrites used by the inferencer are equivalences under the same semantics used to verify Rhodium rules.

Logical simplification rewrites are used to transform a formula into one with a simpler logical form. This includes simplifying conjunctions and disjunctions that contain $true$ or $false$ as clauses, pushing negations into formulas, pushing universal quantifiers over conjunctions and pushing existential quantifiers over disjunctions.

IL simplification rewrites simplify the formula based on properties that are specific to the semantics of the IL. For example, the rule $[*\&]$ says $\sigma(*(\&X)) \rightsquigarrow \sigma(X)$, i.e., taking the address of a variable and then dereferencing gives the value of the variable. Language simplification rewrites can be divided into term rewrites and formula rewrites; the simplifications differ only by the context in which they occur.

Example. The example in Figure 5 uses IL simplification in steps (2), (4), and (5). In step (2) the $[\mathring{=}_\&]$ rule is used to simplify $\sigma(\&X) = \sigma(\&Y)$ to $X \mathring{=} Y$; X and Y have they same address if and only if they are instantiated to the same program variable. The rule $[\mathring{\neq}_\&]$ is used similarly. The rule $[\mathring{=}_c]$ simplifies $\sigma(K) = \sigma(C)$ to $K \mathring{=} C$; two constants have the same value if and only if they are the same syntactic constant.

Logical simplification is used in steps (4) and (5) of the example. In step (4) a universal quantifier is pushed inside a conjunction. In step (5) universal quantifiers are removed when the quantified formula no longer contains the quantified variable.

Example. The example in Figure 6 uses logical simplification in steps (4) and (5) in the same manner as the previous example.

4.4 Strengthening

The function *Strengthen* is applied when simplification does not remove all references to the program state σ. Strengthening is defined by strengthening rewrites. The most important strengthening rewrites are shown in Figure 14. A strengthening rewrite that rewrites P_{old} to P_{new} can occur in a positive term hole if P_{new} implies P_{old} (written $P_{old} \rightsquigarrow_{S^+} P_{new}$). The rewrite can occur in a negative term hole if P_{old} implies P_{new} (written $P_{old} \rightsquigarrow_{S^-} P_{new}$).

The *Strengthen* function tries to apply the strengthening rewrites found in Figure 14. Each strengthening can be thought of as a different approximation of information known at run time. A call to *Strengthen* computes the seat of all possible strengthening rewrites of its argument and returns their disjunction. Formally, this can be written

$$Strengthen(\phi, decls) = \bigvee \left\{ \begin{array}{ll} \phi' | & (\phi = \mathcal{F}^+[P_{old}] \wedge P_{old} \rightsquigarrow_{S^+} P_{new} \wedge \phi' = \mathcal{F}^+[P_{new}]) \\ & \vee(\phi = \mathcal{F}^-[P_{old}] \wedge P_{old} \rightsquigarrow_{S^-} P_{new} \wedge \phi' = \mathcal{F}^-[P_{new}]) \end{array} \right\}$$

[6]A positive position of a formula is one under an even number of negations. A negative term hole is under an odd number of negations.

Context Rules

$\mathcal{F}^t[\mathcal{T}_1] \rightsquigarrow \mathcal{F}^t[\mathcal{T}_2]$ if $\mathcal{T}_1 \rightsquigarrow \mathcal{T}_2$
$\mathcal{F}[\mathcal{F}_1] \rightsquigarrow \mathcal{F}[\mathcal{F}_2]$ if $\mathcal{F}_2 \rightsquigarrow \mathcal{F}_2$

Logical Simplifications

$[logSimp]$
$P \wedge true \rightsquigarrow P$
$P \wedge false \rightsquigarrow false$
$P \vee true \rightsquigarrow true$
$P \vee false \rightsquigarrow P$
$\neg(P \wedge Q) \rightsquigarrow \neg P \vee \neg Q$
$\neg(P \vee Q) \rightsquigarrow \neg P \wedge \neg Q$
$\forall X.(P \wedge Q) \rightsquigarrow \forall X.(P) \wedge \forall X.(Q)$
$\exists X.(P \vee Q) \rightsquigarrow \exists X.(P) \vee \exists X.(Q)$
$\forall X.(P) \rightsquigarrow P(X$ not free in $P)$
$\exists X.(P) \rightsquigarrow P(X$ not free in $P)$

IL Simplification

Term Rewrite Rules

$[*\&]$ $\sigma(*(\&X)) \rightsquigarrow_T \sigma(X)$

Formula Rewrite Rules

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

Figure 13: Simplification rules

Context Rules

$$F^+[F_1] \rightsquigarrow_{S^+} F^+[F_2] \;\text{ if }\; F_1 \rightsquigarrow_{S^+} F_2$$
$$F^-[F_1] \rightsquigarrow_{S^-} F^-[F_2] \;\text{ if }\; F_1 \rightsquigarrow_{S^-} F_2$$

Strengthening Rules

$[\textit{syntactic}]$	$\sigma(T_1) = \sigma(T_2) \rightsquigarrow_{S^+} T_1 \doteq T_2$
$[op_{expand}]$	$F^t[\sigma(\ op\ \ T_1 \ldots T_n)] \rightsquigarrow_{S^+}$
	$(\bigwedge_{i=1 \ldots n} \sigma(C_i) = \sigma(T_i)) \wedge F^t[eval(\ op\ \ C_1 \ldots C_n)]$
$[\textit{quant swap}]$	$\forall X.\exists Y.P(X,Y) \rightsquigarrow_{S^+} \exists Y.\forall X.P(X,Y)$
$[\forall\ \textit{case}]$	$\forall X.\big[F_1 \vee F_2\big] \rightsquigarrow_{S^+} \forall X.\big[F_1\big] \vee \forall X.\big[F_2\big]$
$[\forall\ \textit{resolution}]$	$\forall X.\big[(F_1 \wedge F_2) \vee (\neg F_1 \wedge F_3)\big] \rightsquigarrow_{S^+}$
	$\big(\forall X.\big[F_1 \wedge F_2\big]\big) \vee \big(\forall X.\big[\neg F_1 \wedge F_3\big]\big) \vee \big(\forall X.\big[F_2 \wedge F_3\big]\big)$
$[\textit{Match}]$	$M \rightsquigarrow_{S^+} F$(one rewrite for each fact schema)

Figure 14: Strengthening Rules

Strengthen returns the disjunction of all possible strengthens because each strengthening loses precisions in a different way. If no strengthening rewrites apply, the result is the disjunction of zero formula, i.e., *false*. Strengthening the formula may

The [*syntactic*] rule allows run-time value equality to be approximated by compile-time syntactic equality. The $[op_{expand}]$ rule expands operators. It approximates the knowledge that each term in the operator expression must be equal to some constant equal by forcing the constants all to be known at compile time.

The rule [*quant swap*] pushes a universal quantifier into an existential quantifier. The rewrite approximates the ability to choose a different Y for each possible X by forcing the choice of one particular Y that must be valid with all X. For example, suppose that X is the program state σ and Y is some run-time value. Rewriting $\forall\sigma.\exists Y.P(\sigma,Y)$ to $\exists X.\forall\sigma.P(\sigma,Y)$ approximates the knowledge that for every run-time program state there is a Y such that $P(\sigma,Y)$ holds with the knowledge that there is some single Y for which $P(\sigma,Y)$ holds for every program state.

The [$\forall$ *case*] rule pushes in a universal quantifier over a disjunction. The rewrite approximates knowledge that one alternative or another always happens with knowledge that one alternative must always happen or the other must always happen. [$\forall$ *resolution*] is a refinement of the [*case*] rule. Resolution captures the fact that $(F_1 \wedge F_2) \vee (\neg F_1 \vee F_3)$ is equivalent to $(F_1 \wedge F_2) \vee (\neg F_1 \vee F_3) \vee (F_2 \wedge F_3)$. The equivalence does not simplify the formula, so it is not useful to apply on its own. However, it is useful to apply resolution just before pushing in a universal quantifier so that more information is retained when the quantifier is pushed in. The [$\forall$ *resolution*] rule combines these two steps. Only one of [$\forall$ *resolution*] or [$\forall$ *case*] is applied on a particular subformula; if [$\forall$ *resolution*] is known to not give any more precision than [$\forall$ *case*], then [$\forall$ *case*] is used.

Example. The example in Figure 5 uses the [$\forall$ *case*] in step (3) to push the universally quantified σ over the disjunction. In this example, the [$\forall$ *case*] rule is sufficient. Since it is always possible to determine at compile time whether two variables are syntactically the same, adding the third case gives no added precision.

Example. The example in Figure 6 uses the [$\forall$ *resolution*] strengthening in step (3). In this example, the [$\forall$ *case*] rule would not have lost information. The sub-formulas $\forall\sigma_{in}.\big[\sigma(\&X) = \sigma(Y)\big]$ and $\forall\sigma_{in}.\big[\sigma(\&X) \neq \sigma(Y)\big]$ do not cover all possible cases; X may point to Y in some valid program states but not others, or the dataflow analysis may be unable to determine whether X points to Y. The third case produced by resolution gives a condition for propagating $\textit{hasConstValue}(X,C)@\textit{out}$ even if it is not known whether X points to Y. The case introduced by resolution says that if $\sigma_{in}(Z)$ and $\sigma_{in}(X)$ both equal $\sigma_{in}(C)$, then $\sigma_{out}(X)$ must equal $\sigma_{out}(C)$. If Y points to X, then $\sigma_{out}(X)$ will hold the $\sigma_{in}(Z)$, i.e., $\sigma_{in}(C)$. If Y does not point to X, $\sigma_{out}(X)$ will hold what it was before, also $\sigma_{in}(C)$.

4.4.1 Matching

A matching rewrite is a strengthening rewrite that replaces a fact meaning with an edge predicates. Matching is a strengthening, because the presence of a fact on an edge implies that the meaning of the fact holds on that edge at runtime, but the meaning of a fact holding on an edge at runtime does not imply that the fact can be guaranteed at compile time to always hold on an edge.

Matching rewrites are different from other rewrites, because they are not a fixed part of the inferencer. The [*Match*] strengthening in Figure 14 is really a pattern that represents many strengthening rewrites. There is one matching rewrite for each edge fact schema given by the optimization writer. For example, the edge fact schema $\mathit{hasConstValue}(X, C)$ has meaning $\forall\sigma \in \Sigma_{ed}.\sigma(X) = \sigma(C)$. When the algorithm is inferring rules for this edge fact schema, one of the matching rewrites will be $\forall\sigma \in \Sigma_{ed}.\sigma(X) = \sigma(C) \rightsquigarrow_{S+} \mathit{hasConstValue}(X, C)$. As with other strengthening rewrites, the left-hand side of the rule must syntactically match the subformula being replaced. To make syntactic matching easier, the meaning is simplified using the *Simplify* function described in section 4.3.

Example. Step (5) in Figure 5 shows matching. Two instances of $\forall\sigma_{in}.\big[\sigma_{in}(X) = \sigma_{in}(C)\big]$ are replaced with $hasConstValue(X, C)@in$.

Example. Step (5) in Figure 6 shows another instance of fact matching. As before, two instances of $\forall\sigma_{in}.\big[\sigma_{in}(X) = \sigma_{in}(C)\big]$ are matched to $\mathit{hasConstValue}(X, C)@\mathit{in}$. Two instances of $\forall\sigma_{in}.\big[\sigma_{in}(Z) = \sigma_{in}(C)\big]$ are matched to $\mathit{hasConstValue}(Z, C)@\mathit{in}$. Finally, $\forall\sigma_{in}.\big[\sigma_{in}(\&X) = \sigma_{in}(Y)\big]$ and $\forall\sigma_{in}.\big[\sigma_{in}(\&X) \neq \sigma_{in}(Y)\big]$ become $\mathit{mustPointTo}(Y, X)@\mathit{in}$ and $\mathit{mustNotPointTo}(Y, X)@\mathit{in}$, respectively.

Finding the correct facts to match is simple in these two examples; in general, matching may be difficult. Many factors introduce difficulty. The formula may contain a subformula equivalent to a fact meaning but syntactically different. To partially remedy this, the meanings for all facts schema meanings are simplified at the beginning of the algorithm. Even with this, syntactic matching is brittle. Fact schema meanings can be arbitrary logical formulas and may be difficult to match.

Consider a fact schema with meaning $\forall\sigma \in \Sigma_{ed}.\sigma(X) = \sigma(C) \wedge \sigma(Y) = \sigma(C)$. This meaning says that the variables X and Y have the same value as the constant C. Suppose the algorithm was trying to match this to the subformula $\forall\sigma \in \Sigma_{ed}.\sigma(X) = \sigma(Y) \wedge \sigma(X) = \sigma(C)$. This subformula has the same semantic meaning but does not match the syntactic meaning; both of the equalities in the meaning have a Rhodium variable and a Rhodium constant, but in the subformula one equality has a Rhodium variable and a Rhodium constant and the other has two Rhodium variables. In practice, most of our fact schema meanings have been single term equalities or inequalities sometimes wrapped in a single quantifier, so syntactic matching often works.

There may be multiple ways to match a subformula to a fact. The choice will affect how well the inferred rules cover all cases. Instead of choosing just one matching fact, the *Match* function returns the disjunction of all facts that match the formula.

5 Implementation

Rhodium is implemented as part of the Whirlwind compiler. Our inference algorithm extends the Rhodium implementation. To infer rules, a user of Whirlwind specifies a file containing edge fact schemas and transformation rules. The system then infers propagation rules for those facts. All matching rewrite is created for each edge fact schema.

The implementation diverges from the description of the algorithm in one significant way. In our algorithm, strengthening rewrites may occur anywhere in the formula, and we try all rewrites on the formula. Combining these can cause a blowup in the size of the formula without adding more precision to the results. To reduce this effect, we have a heuristic for choosing where to strengthen a formula. The algorithm searches the formula for the program state σ. When σ is found, we look for the nearest quantifier over σ (the program state is always quantified). Strengthening is performed *only* on those quantified formulas. This heuristic reduces the blowup. However, it may cause our algorithm to lose some precision, because it is sometimes useful to strengthen larger formulas.

6 Results

We inferred rules for 10 edge fact schemas. The schemas included variable-to-expression equality and disequality, mathematical inequalities, and facts for determining which variables have been declared and the runtime types of variables.

For the edge fact schemas described above and fourteen IL statements, the implementation can infer rules in under 10 minutes on a modern machine. The output contains 6186 propagation rules (once disjunctions are broken into multiple rules). These rules have all been verified with the Rhodium verifier. 69 rules do not pass the "sanity test", which tests whether a rule's premise could ever be satisfied. For example, the premise of an inferred rule may constrain some variable to be equal to both a constant and an array. These rules are not useful but will never cause the resulting dataflow analysis to be incorrect.

It is unclear how useful these rules are in practice. One way to evaluate the rules would be to use them to analyze benchmark programs and measure the speed-up of the optimized programs. However, our execution engine for Rhodium is currently too slow to optimize any realistic programs with such a large number of rules.

Another way to evaluate the rules is to compare the information they compute with information computed by a set of rules we believe to be good. This is the approach we take. We compare our inferred rules to a set rules handwritten by an expert in our research group. The handwritten rules consist of 116 rules for propagating 10 dataflow facts to support 15 code transformations. The dataflow facts are the same facts used in inference. The handwritten rules do not use any quantifiers. Handwritten Rhodium rules may propagate a fact for multiple statements and multiple types of expressions. When the handwritten rules are expanded so that each rule propagates a fact for only one statement form and one expression form per expression variable, there are 613 expanded handwritten rules. Section 6.1 explains how we compare two sets of rules.

Sections 6.2 and 6.3 show the results of comparing the handwritten rules and the inferred rules. Both sets of rules propagate some information that the other set does not propagate. However, since both the handwritten rules and inferred rules are written in Rhodium, the two sets of rules can be combined to get better results than either set alone.

6.1 Comparison method

Our goal when comparing two sets of rules is to see which rules in the first set are subsumed by some combination of rules in the second set. A rule r is said to be subsumed by a set of rules R if whenever r propagates a fact, some combination of rules in R propagates the same fact under the same conditions. A rule propagates a fact whenever there exists an instantiation of its metavariables that make its antecedent true. Thus, a rule is subsumed if, for any instantiation that makes its antecedent true, there is true some rule in R has a true antecedent under the same instantiation. Because there can be different instantiations of r that make its antecedent true, it may take multiple rules in R to subsume all instantiations. This gives our method for checking to see if a rule is subsumed. To see if a rule r is subsumed by some set of rules R, use a theorem prover to see if the antecedent of r implies the disjunction of the antecedents of the rules in R, taking care to match variable names properly.

This algorithm allows the comparison of the handwritten rules and the inferred rules. First, we will test to see which handwritten rules are subsumed by inferred rules. If a handwritten rule is subsumed by inferred rules, then our algorithm was able to perform as well as a human for that particular rule. If a handwritten rule is not subsumed by inferred rules, then there are potentially some situations under which that handwritten rule can propagate information our inferred rule does not.

Our second test determines which inferred rules are subsumed by handwritten rules. If a inferred rule is subsumed by a handwritten rule, then that rule provides no more information than the handwritten rules. If an inferred rule is not subsumed by a handwritten rule, then the inferred rule propagates information in a situation where the handwritten rules potentially did not.

In both tests, rules that are not subsumed are only potentially novel. The theorem prover may be unable to prove the implication even though it is true. Another, more subtle reason is inherent of the nature of our test. For each rule, our comparison tests whether there is some rule in the other set that propagates the

Category	Number	Percent
Subsumed	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
Requires mathematical reasoning	2	0.3
Total:	613	100.0

Figure 15: Result of testing if handwritten rules are subsumed by inferred rules

same information under the exact same conditions. However, it may be that the other set of rules propagates the information under more general conditions. We will show an example of this in section 6.3.

6.2 Comparing handwritten rules to inferred rules

Our inference method is able to infer rules which subsume 472 (77%) of these expanded rules. Table 16 shows our results, including the different categories of failures. Each entry in the table has the number of rules in that category and the percentage of the total handwritten rules that fall into that category. These categories of failures will be discussed in more detail in the rest of this section.

6.2.1 Introduces new variable or constant

Our inference algorithm never introduces new variables and rarely introduces new constants. Many of the handwritten rules do. For example, the rule

$$\begin{array}{l} \textbf{\textit{if}}\ \ \mathit{currStmt} \mathrel{\dot{=}} [\mathit{skip}] \wedge \\ \quad \mathit{varEqualsExpr}(X, Y)@\mathit{in} \wedge \mathit{varEqualsExpr}(Y, E)@\mathit{in} \\ \textbf{\textit{then}}\ \ \mathit{varEqualsExpr}(X, E)@\mathit{out} \end{array}$$

is not subsumed by the inferred rule. This rule introduces a variable, Y, mentioned in neither the statement (which mentions no variables) nor the consequent (which mentions only X and E).

Determining heuristics for introducing variables and constants that were not mentioned in the original predicate or statement is difficult to fit into a pure rewrite system. For example, it would be easy to add some new variables with the rewrite $\sigma(X) = \sigma(Y) \leadsto_{S+} \sigma(X) = \sigma(Z) \wedge \sigma(Z) = \sigma(Y)$, but without extra constraints on the use of this rule, there is no bound to the number of new variables that may be added. Such an algorithm would have to have some technique for limiting the number of new variables and constants introduced. For example, the rewrite above could be constrained to only add new variables if both X and Y are variables from the original statement or the postcondition. If we consider only equalities, another way of increasing coverage would be to guarantee that for all variables X, Y, and Z, if the dataflow analysis defined by our rules finds $\sigma(X) = \sigma(Z)$ and $\sigma(Y) = \sigma(Z)$ it will also find $\sigma(X) = \sigma(Z)$. That is, for all variables know to be in the same congruence class, our dataflow analysis could guarantee that an edge predicate is propagate for every pair from that set. A technique that would be useful for the special case of

6.2.2 Weakness of purely syntactic reasoning

Doing rewrites only on subformulas that syntactically match the left-hand-side of a rewrite rule decreases the precision of the rules we infer. Some handwritten rules are not subsumed because the resolution strengthening rewrite is not always used. Sometimes, the formula the algorithm is trying to strengthen does not have the exact form looked for by the resolution strengthening rewrite, but it is equivalent to a formula in that form. When that happens, our results lose precision.

Another limitation of our treatment of the formula is that the heuristic described in section 5 for deciding where to apply strengthening is too fine grained for some matches. This is particularly problematic when the optimization writer defines a fact with a conjunction in the meaning. Consider the fact

$$\begin{array}{l}\textbf{define edge fact schema } \mathit{isArrayFact}(X : \mathit{Var}, I : \mathit{Var}) \\ \quad \textbf{with meaning } \mathit{isArray}(\sigma(X)) \wedge \sigma(I) = \mathit{arrayLength}(\sigma(X))\end{array}$$

The meaning of $\mathit{isArrayFact}(X, I)$ is that X is an array and X has length I. The meaning of this fact is represented internally as

$$(\forall \sigma \in \Sigma_{ed}.\mathit{isArray}(\sigma(X))) \wedge (\forall \sigma \in \Sigma_{ed}.\sigma(I) = \mathit{arrayLength}(\sigma(X)))$$

To match this meaning, matching would need to be done on the conjunctions. However, our heuristic only matches on the quantifiers. Because of this, we never infer facts that use the $\mathit{isArrayFact}(X, I)$ edge predicate.

6.2.3 Lost information

Sometimes the algorithm infers rules that are too strict because it does not utilize all of the information available to it. For example, after a branch statement, it is known that the branch guard is *true* on the $\mathit{out}[0]$ branch and *false* on the $\mathit{out}[1]$ branch. The following handwritten rule uses this property.

$$\begin{array}{l}\textbf{\textit{if}} \;\; \mathit{currStmt} \triangleq [\text{if } X \text{ then goto } L_1 \text{ else } L_2] \\ \textbf{\textit{then}} \;\; \mathit{varEqualsExpr}(X, \mathit{true})@\mathit{out}[0]\end{array}$$

This rule is never inferred because the inference algorithm never considers that the variable in the edge predicate on the edge $\mathit{out}[0]$ could be the same as the branch guard.

The inference algorithm also loses information because it does not keep track of variable types. In a rule for the array declaration statement [decl $X[I]$], X must be an array and cannot be equal to a constant, but the inference algorithm does not take advantage of that information.

6.2.4 Missing strengthening rewrites

The inference algorithm cannot detect whether or not it has been in a particular state before. Because of this, the set of strengthening rewrite rules cannot include rewrite rules that may put the inferencer into a state it has been in before. For example, the algorithm can only include one of the two strengthening rewrite rules $\sigma(T_1) = \sigma(T_2) \rightsquigarrow_{S+} \sigma(T_1 \leq T_2) = \sigma(\mathit{true}) \wedge \sigma(T_1 \geq T_2) = \sigma(\mathit{true})$ and $\sigma(T_1 \leq T_2) = \sigma(\mathit{true}) \rightsquigarrow_{S+} \sigma(T_1) = \sigma(T_2)$. We can put only one of the two strengthening rewrite rules into the inference algorithm and lose rules that would be inferred if the other were used.

6.2.5 Missing mathematical reasoning

Our system tries to infer propagation rules for an edge fact schema used to determine the induction variable of a loop. That edge fact schema has the meaning $\forall \sigma \in \Sigma_{ed}.\sigma(X) = \sigma((Y - C) \; op \; Z)$ (op can be any binary operator). The handwritten rules for this fact involve some mathematical reasoning that would be complicated to express as normalizations and strengthening rewrite rules.

It may be possible to encode a large number of mathematical rules as rewrite rules. However, this approach does not really capture the structure of the mathematical relationships. A more general approach might be to integrate the inference algorithm with a system that can do more general mathematical reasoning.

6.3 Comparing inferred rules to handwritten rules

The inference algorithm generates 6186 rules. Our comparison method finds that 1379 of these rules are subsumed by the handwritten rules. 69 of these rules can never have their antecedent satisfied and do not

Category	Number	Percent
Subsumed	1379	22.2
Trivially not subsumed	69	1.1
Potentially novel	4738	76.6
Total:	6186	100.0

Figure 16: Result of testing if inferred rules are subsumed by handwritten rules.

increase the expressive power of the rule set. This leaves 4738 rules which are potentially useful and novel. These rules may not all give useful additional information for guiding compiler optimizations.

There are too many potentially novel rules to analyze them all. Instead, we will give an example of an inferred rule that really is novel and an example of one that gives no useful information. The rule below is novel.

$$\begin{array}{l} \textbf{\textit{if}}\ \ \mathit{currStmt} \mathrel{\dot{=}} [*Z := Y] \wedge \\ \quad \mathit{varEqualsExpr}(X, W)@\mathit{in} \wedge \mathit{varEqualsExpr}(Y, W)@\mathit{in} \wedge \mathit{mustNotPointTo}(Z, W)@\mathit{in} \\ \textbf{\textit{then}}\ \ \mathit{varEqualsExpr}(X, W)@\mathit{out} \end{array}$$

This rule says that if X and Y both equal W before Y is assigned to $*Z$ and the assignment does not change W, then X will equal W afterward. This rule allows the information that X equals W to be preserved without knowing whether or not Z points to X. Although this is a syntactically simple rule, pointers make it difficult to reason about. In this case, our algorithm has inferred a rule that was not expressed in the handwritten rules.

Some rules are not really novel. The rule

$$\begin{array}{l} \textbf{\textit{if}}\ \ \mathit{currStmt} \mathrel{\dot{=}} [Z := K] \wedge \\ \quad Y \not\mathrel{\dot{=}} Z \wedge X \not\mathrel{\dot{=}} Z \wedge \mathit{varEqualsExpr}(Y, X)@\mathit{in} \\ \textbf{\textit{then}}\ \ \mathit{varEqualsExpr}(X, Y)@\mathit{out} \end{array}$$

is not novel. The rule says that the *varEqualsExpr* edge predicate is symmetric as long as its arguments are unchanged. The handwritten rules do not have a rule that say this is true for an the particular statement $[Z := K]$. However, it does have a rule that ensures that whenever $\mathit{varEqualsExpr}(Y, X)$ is on an edge, $\mathit{varEqualsExpr}(X, Y)$ will be there too. Therefore, the rule above is novel by the way we measure novelty, but does not give information the handwritten rules do not give.

Without using the potentially novel rules to optimize real code, we cannot to tell whether they are useful or obscure corner cases that never come up in practice. However, even without knowing what portion of rules are useful, the number of potentially novel rules points out that humans are not good at finding all the corner cases for a dataflow analysis; our inference algorithm helps to cover such cases.

Our algorithm also helps to cover some mundane rules. For every dataflow fact and every statement, there must be a rule that says that if an edge predicate is true on the edge into a CFG node and the node does not change the arguments of the edge predicate, then the edge predicate is known after the CFG node; our algorithm infers these rules.

Finally, our algorithm removes the burden of reasoning about interactions among different dataflow facts. In our experience, when a user adds a new dataflow fact to a set of handwritten rules, he will write rules for that new fact that take advantage of the facts already defined, but they will not go back and write propagation rules for the old facts that exploit the new fact. However, by rerunning the inferencer, rules that take advantage of interactions among all of the facts are generated.

7 Future Work

Short term future work can be divided into two broad categories: increase the coverage of the inferred rules and find better metrics for evaluating inferred rules. In the long run, we would like to further decrease the work involved in writing compiler optimizations by inferring edge fact schemas and, eventually, code transformations.

The evaluation in section 6.2 points out several weaknesses of our algorithm. Fixing those weaknesses would be one way to increase coverage. Another way to increase coverage would be to find a more systematic method of adding simplification and strengthening rewrite rules. Currently rewrite rules are added to the inference algorithm as we see a use for them; this makes it difficult to know what may be missing from the rewrite rules.

The other area of future work is to use different metrics to assess the quality of the rules we infer. The best measure of the quality of the rules is how well they enable optimizations. Once we have an efficient execution engine for Rhodium rules, we can compile benchmark programs using the handwritten rules or the inferred rules and compared the speed-up in each case.

8 Related Work

Our work is closely related to predicate abstraction [7, 5, 2, 11]. The domain in predicate abstraction consists of a fixed, finite cartesian product of boolean values, where each boolean value is the abstraction of a predicate over concrete states. Because such domains are finite, it is possible to infer the flow functions by asking a theorem prover to try all possible abstract transitions. The generated flow functions consist of those transitions that the theorem prover was able to validate. We could use this approach; our predicates would be edge predicates over specific metavariables. For example, one predicate would be $\mathit{hasConstValue}(X, C)@\mathit{in}$ and another would be $\mathit{hasConstValue}(Y, C)@\mathit{in}$. However, this approach does not scale to our domain. Predicate abstraction is usually used when the number of predicates is small; because each possible metavariable instantiation of an edge predicate is a separate predicate, this approach does not scale to our domain.

The recent work of Reps, Sagiv, and Yorsh [18, 20] 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 (this includes $\mathit{hasConstValue}(X, C)$). 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 that we do not guarantee. However, the flow function of Reps *et al.* is not specialized to a particular domain: they describe one single flow function that works for all finite-height domains. 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 so our generated flow functions can be expressed as simple rules that do only 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. The work of Reps *et al.* is useful because of the theoretical guarantees it provides but does not address our goal of inferring flow functions that can be computed once and used on any program.

Another system that infers flow functions automatically is the TVLA system [13]. 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 TVLA system runs an interpretation algorithm in the forward direction on every call to the flow function. Our system, on the other hand, pre-computes much of the flow function when it is generated, resulting in simple flow functions that can be evaluated using syntactic checks.

Our work is also related to the HOIST system for automatically deriving static analyzers for embedded systems [17]. 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.

The search that we do with 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 [14], NuPRL [4], Twelf [16, 19], the Boyer-Moore theorem prover [9, 10], Isabelle [15], and HOL [6]. Users of these systems can use tactics (or variations thereof) to guide the theorem prover in its 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.

Another proof-search technique closely related to our algorithm 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 anymore, 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.

9 Conclusion

We have developed an algorithm for inferring sound dataflow analyses in the Rhodium language given just the semantic meaning of the dataflow facts. This algorithm is able to infer rules that subsume 77% of hand written rules and infers rules that are novel. The inference tool provides a useful tool for aiding an optimization writer in writing dataflow analyses. It takes care of many of the tedious cases for writing dataflow rules and covers cases a human might have missed. Finally, even without perfect performance, inferring rules still provides benefits to an optimization writer; the inferred rules use Rhodium syntax, so they can be combined with handwritten rules to get a better set of rules.

References

[1] J.M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, pages 297–347, 1992.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of ACM SIGPLAN PLDI'2001*, Snowbird, Utah, USA, June 2001.

[3] Robert Cartwrigth and Derek Oppen. The logic of aliasing. Technical Report STAND-CS-79-740, Standform, 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, NJ, 1986.

[5] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Proceedings of the 11th International CAV Conference*, June 1999.

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

[7] Susanne Graf and Hassen Saidi. Construction of abstract state graphs of infinite systems with PVS. In *Proceedings of the 9th International CAV Conference*, June 1997.

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

[9] 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.

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

[11] Shuvendu Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, Edinburgh, UK, July 2005.

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

[13] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proceedings of the Seventh International Static Analysis Symposium (SAS '00)*, Santa Barbara, California, USA, June 2000.

[14] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In M. A. McRobbie and J.K. Slaney, editors, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[15] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecure Notes in Computer Science*. Springer Verlag, 1994.

[16] F. Pfenning and C. Schurmann. Sytsem description: Twelf – a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer-Verlag, July 1999.

[17] John Regehr and Alastair Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Proceedings of the 11th International ASPLOS Conference*, Boston, Massachusetts, USA, October 2004.

[18] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Proceedings of VMCAI 2004*, Venice, Italy, January 2004.

[19] C. Schurmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner H. Kirchner, editor, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Computer Science*, pages 286–300. Springer-Verlag, July 1998.

[20] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proceedings of TACAS'2004*, Barcelona, Spain, March 2004.