Automatically Inferring Sound Dataflow Functions from Dataflow Fact Schemas

Erika Rice [1] Sorin Lerner [2] Craig Chambers [3]

Department of Computer Science & Engineering
University of Washington
Seattle, WA, USA

Abstract

In previous work, we presented a language called Rhodium for writing program analyses and transformations that could be checked for soundness automatically. In this work, we present an algorithm for automatically *inferring* sound flow functions given only a set of dataflow fact schemas. By generating the flow functions mechanically, our approach reduces the burden on compiler writers. This paper presents a detailed description of our algorithm and shows how it works on several examples. We have run our algorithm by hand on all the statements of a simple C-like intermediate language for an is-constant fact schema, a points-to fact schema, and a variable-equality fact schema. Our algorithm generated a total of 71 rules for these cases. It generated all but one of the rules we had written by hand for these dataflow fact schemas, and it also generated new useful rules that we had not thought of previously.

Key words: Compiler optimization, dataflow analyses, generated flow functions, automated correctness proofs.

1 Introduction

Compilers are an important part of the infrastructure relied upon by developers. If the compiler is buggy, then so are potentially all applications compiled with it. Our broad research agenda is to provide better support for writing sound compiler optimizations. In previous work [5,4], we presented a language called Rhodium for writing compiler optimizations that could be checked for soundness automatically. Programmers would declare dataflow fact schemas with an associated semantic meaning and then would write flow functions for propagating facts that are instances of these schemas as well as rules for performing program transformations. Once written in Rhodium, these flow

[1] Email: erice@cs.washington.edu
[2] Email: lerns@cs.washington.edu
[3] Email: chambers@cs.washington.edu

functions and these program-transformation rules could be proven sound automatically.

In this paper, we present a technique for automatically *inferring* sound forward dataflow functions given only the declarations of the dataflow fact schemas including their semantic meanings. As a result, the programmer need not be burdened with writing the flow functions any more, making it even easier to produce sound dataflow analyses.

Flow functions in Rhodium are expressed using *propagation rules*, which state under what conditions a fact can be introduced on the outgoing edge of a CFG node. Our approach for generating forward propagation rules for a fact schema works backward from its semantic meaning. The semantic meaning gives us the requirement that must hold for an instance of the fact schema to be introduced on the outgoing edge of a node. Weakest preconditions are used to compute the condition that must hold *before* the node for the soundness requirement to hold *after* the node. Finally, we re-express this weakest precondition in terms of only the semantic meanings of fact schemas that have been given to us.

This paper presents a detailed description of the above algorithm, and shows how it works on several examples. We have run our algorithm by hand on all the statements of a simple C-like intermediate language for an is-constant fact schema, a points-to fact schema, and a variable-equality fact schema. Our algorithm generated a total of 71 propagation rules for these fact schemas. It generated all but one of the rules we had written by hand for these schemas, and it also generated new useful rules that we had not thought of.

Section 2 presents background material on how dataflow analyses are written in Rhodium and how they are checked for soundness automatically. Section 3 presents an overview of our approach, and then section 4 presents the details of our algorithm. Section 5 presents our plans for current and future work, and, finally, Section 6 presents related work.

2 Background on Rhodium

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 flow functions, we will consider a simpler IL, shown in Figure 1, that does not have functions calls or arrays. The IL program is represented as a CFG with each node representing a simple register-transfer-level statement.

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

Stmts	s ::=	`decl` $x := e$ | `skip` | $lhs := e$ | $x :=$ `new` | `if` b `goto` ι `else` ι
Exprs	e ::=	$b \mid *x \mid \&x \mid op\ b\ \ldots\ b$
Locatables	lhs ::=	$x \mid *x$
Base Exprs	b ::=	$x \mid c$
Ops	op ::=	various operators with arity ≥ 1
Vars	x ::=	`x` | `y` | `z` | ...
Consts	c ::=	constants
Indices	ι ::=	$0 \mid 1 \mid 2 \mid \ldots$

Fig. 1. Grammar of the IL

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*** *stmt*$(X := C)$
5. ***then*** *hasConstValue*(X, C)@*out*
6. ***if*** *hasConstValue*(X, C)@*in* $\wedge$ *stmt*$(Z := K) \wedge X \neq Z$
7. ***then*** *hasConstValue*(X, C)@*out*

Fig. 2. Simple constant propagation analysis in Rhodium.

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 2. A fact schema is a parametrized dataflow fact (a pattern, in essence) that can be instantiated to create actual dataflow facts. With the declaration on line 2, 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 program being analyzed, and C ranges over constants.[4]

Propagation rules in Rhodium indicate how edge facts are propagated across CFG nodes. For example, the rule on lines 6-7 of Figure 2 defines a condition for preserving a *hasConstValue* fact across an assignment node: if the fact *hasConstValue*(X, C) appears on the incoming CFG edge of an assignment node and X is not the variable assigned to, then the dataflow fact *hasConstValue*(X, C) should appear on the outgoing edge of n.

The left-hand side of a rule is called the *antecedent* and the right-hand side the *consequent*. Each propagation rule is interpreted within the context of a CFG node. Edge facts are followed by @ signs, with the name after the @ sign indicating the edge on which the fact appears. For example, *hasConstValue*(X, C)@*in* is true if the incoming CFG edge of the current node is annotated with *hasConstValue*(X, Y). For explanatory purposes, we make the simplifying assumption that a node with multiple incoming edges simply merges facts using intersection, and a node with multiple outgoing edges simply propagates the incoming facts on all outgoing edges.

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

The semantics of a propagation rule on a CFG is as follows: for each substitution of the rule's free variables that make the antecedent valid at some node in the CFG, the fact in the consequent is propagated. For the rule described above, the $\mathit{hasConstValue}(X, C)$ 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.

To check Rhodium dataflow analyses for soundness, programmers must specify a semantic meaning for each fact schema, in the form of a predicate over a program state σ. For example, the meaning of $\mathit{hasConstValue}(X, C)$, shown on line 3, 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 $\mathit{hasConstValue}(X, C)$ appears on an edge in the CFG, then for any σ that may occur at run-time when control reaches that edge, [5] $\sigma(X) = \sigma(C)$ holds. Henceforth, when we refer to a fact schema, we refer to the parameterized fact *and* its meaning.

Rhodium dataflow analyses are checked for soundness automatically by discharging a soundness obligation for each propagation rule. For each rule we ask the theorem prover so 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. We have shown by hand that if all the propagation rules in a Rhodium program pass this condition, then the induced dataflow analysis is sound.

3 Overview of our Approach

In our previous work we automatically checked that Rhodium propagation rules were sound. In this work, we automatically *infer* sound forward propagation rules given only the declarations of the dataflow fact schemas. For the example in Figure 2, our goal would be to infer propagation rules such as the ones on lines 4-7 from the definition on lines 2-3.

In the case of *hasConstValue*, our problem reduces to finding formulas ψ that make the propagation rule ***if*** ψ ***then*** $\mathit{hasConstValue}(X, C)@\mathit{out}$ sound. Intuitively, this rule is sound if the meaning of $\mathit{hasConstValue}(X, C)$ holds after a CFG node n whenever the meaning of ψ holds before n. As a result, all the formulas ψ we are looking for must be preconditions of n for establishing $\sigma(X) = \sigma(C)$ after n. To find such preconditions, our approach is to first compute the *weakest* precondition ϕ with respect to n that establishes $\sigma(X) = \sigma(C)$, and then to search through the space of formulas that imply ϕ to find appropriate formulas ψ. In logical terms, the process of finding formulas that imply ϕ is called strengthening.

To compute the weakest precondition of $\sigma(X) = \sigma(C)$ through a node n, we do a case analysis on the form of the statement at node n. Consider for example the case where the statement is of the form $Y := K$, where Y and K range respectively over variables and constants of the IL program. The

[5] The quantification of σ is implicit in the user's declaration.

weakest precondition rule for assignments is as follows:

$$wp(Y := E, \alpha) = \alpha[Y \mapsto E]$$

where $\alpha[Y \mapsto E]$ denotes the formula α with Y replaced by E. 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 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 expands to $\sigma(X) = \sigma(C)$. The weakest precondition ϕ is then as follows (we use the symbol $\doteq$ for syntactic equality of ASTs and $=$ for semantic equality of run-time values):

$$\begin{aligned} \phi &= (X \doteq Y \wedge \sigma(K) = \sigma(C)) \vee (X \not\doteq Y \wedge \sigma(X) = \sigma(C)) \\ &= (X \doteq Y \wedge K \doteq C) \vee (X \not\doteq Y \wedge \sigma(X) = \sigma(C)) \\ &\qquad\qquad \text{since } K \text{ and } C \text{ are constants } \sigma(K) = \sigma(C) \Leftrightarrow K \doteq C \end{aligned}$$

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 refer to σ. Our task then is to map all the σ's in ϕ using dataflow facts. In the constant propagation example, the user would provide the dataflow fact schema $\mathit{hasConstValue}(X, C)$ with meaning $\sigma(X) = \sigma(C)$. By searching for syntactic matches of the form $\sigma(X) = \sigma(C)$ in ϕ and replacing them with $\mathit{hasConstValue}(X, C)@\mathit{in}$, we get:

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

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

if $\mathit{stmt}(Y := K) \wedge \psi$
then $\mathit{hasConstValue}(X, C)@\mathit{out}$

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

if $\mathit{stmt}(Y := K) \wedge X \doteq Y \wedge K \doteq C$
then $\mathit{hasConstValue}(X, C)@\mathit{out}$

if $\mathit{stmt}(Y := K) \wedge X \not\doteq Y \wedge \mathit{hasConstValue}(X, C)@\mathit{in}$
then $\mathit{hasConstValue}(X, C)@\mathit{out}$

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

function *GenerateRules*(*decls*: *set*[*FactSchemaDecl*]): *set*[*Rule*]
  let $results := \emptyset$
  for each "***define edge fact schema*** F ***with meaning*** M" $\in decls$ **do**
    for each statement form S **do**
      let $\phi := wp(S, M)$
      let $\psi := PerformBackwardSearch(\phi, decls)$
      if $\psi \neq false$ **then**
        let $rule :=$ "***if*** $stmt(S) \wedge \psi$ ***then*** $F@out$"
        $results := results \cup \{rule\}$
  return *results*

function *PerformBackwardSearch*(ϕ: *Formula*, *decls*: *set*[*FactSchemaDecl*]): *Formula*
  let $\phi_{norm} := Normalize(\phi)$
  let $\phi_{mapped} := MapFacts(\phi_{norm}, decls)$
  if ϕ_{mapped} contains no σ **then**
    return ϕ_{mapped}
  let $\phi_{stren} := Strengthen(\phi_{mapped})$
  return $PerformBackwardSearch(\phi_{stren}, decls)$

Fig. 3. Algorithm for generating rules from fact declarations

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) or 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 one such 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 soundness 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 sound.

4 Algorithm for Inferring Rules from Facts

Our algorithm for generating rules is shown in Figure 3. For explanatory purposes, we only present the algorithm that handles single-input-single-output nodes. [6] For each fact schema declared by the user and for each statement

[6] For nodes with many incoming and outgoing edges, we would run the algorithm from Figure 3 on each input-output edge pair. In this case, *wp* would take input and output edges as additional parameters.

$$
\begin{array}{rcl}
 & (X \doteq Y \wedge K \doteq C)\ \vee (X \not\doteq Y \wedge \mathit{hasConstValue}(X,C)@\mathit{in}) \vee & \\
 & (K \doteq C \wedge \mathit{hasConstValue}(X,C)@\mathit{in}) & \\
\hline
(4) & & \mathit{fact\ match}, \mathit{logNorm} \\
 & (\forall\sigma.\big[X \doteq Y\big] \wedge \forall\sigma.\big[K \doteq C\big])\ \vee (\forall\sigma.\big[X \not\doteq Y\big] \wedge \forall\sigma.\big[\sigma(X) = \sigma(C)\big]) \vee & \\
 & (\forall\sigma.\big[K \doteq C\big] \wedge \forall\sigma.\big[\sigma(X) = \sigma(C)\big]) & \\
\hline
(3) & & \mathit{logNorm} \\
 & \forall\sigma.\big[X \doteq Y \wedge K \doteq C\big]\ \vee \forall\sigma.\big[X \not\doteq Y \wedge \sigma(X) = \sigma(C)\big]\ \vee \forall\sigma.\big[K \doteq C \wedge \sigma(X) = \sigma(C)\big] & \\
\hline
(2) & & \forall\ \mathit{resolution} \\
 & \forall\sigma.\big[(X \doteq Y \wedge K \doteq C)\ \vee (X \not\doteq Y \wedge \sigma(X) = \sigma(C))\big] & \\
\hline
(1) & & \doteq_{I_{\&}}, \not\doteq_{I_{\&}}, \doteq_{I_c} \\
 & \forall\sigma.\big[(\sigma(\&X) = \sigma(\&Y) \wedge \sigma(K) = \sigma(C))\ \vee (\sigma(\&X) \neq \sigma(\&Y) \wedge \sigma(X) = \sigma(C))\big] &
\end{array}
$$

Fig. 4. Inference steps for $\mathit{hasConstValue}(X, C)$ for statements of the form $Y := K$

form, we first compute the weakest precondition of the meaning provided in the declaration with respect to the statement form using the *wp* function (line 4). We then perform a backward search in our inference system starting at 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 normal form using the *Normalize* function (line 10). We then use *MapFacts* (line 11) to map parts of the formula to facts, thus removing some references to σ. If all references to σ have been removed (line 12), then we have found a valid antecedent and the search is over (line 13). If not, we strengthen the formula using *Strengthen* (line 14), and continue the search with the resulting stronger formula (line 15).

The algorithm in Figure 3 depends on the four functions *wp*, *Normalize*, *MapFacts* and *Strengthen*. These four functions are described in more detail in the following four subsections.

Throughout our explanations, we will use the examples in Figures 4 and 5, which show the inference steps used by our algorithm for $\mathit{hasConstValue}(X, C)$ for statements of the forms $Y := K$ and $*Y := Z$. In these examples, the fact schemas available for matching are $\mathit{hasConstValue}(X, C)$ with meaning $\sigma(X) = \sigma(C)$, $\mathit{mustNotPointTo}(X, W)$ with meaning $\sigma(X) \neq \sigma(\&W)$ and $\mathit{mustPointTo}(X, W)$ with meaning $\sigma(X) = \sigma(\&W)$. The examples should be read bottom-up: the bottommost formula is the weakest precondition ϕ as computed by *wp*, 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 deduction. Each backward inference step represents multiple steps taken by *Normalize*, *Strengthen* and/or *MapFacts*. The labels to the right of an inference step identify the rewrite rules used for this step. The collection of all rewrite rules in our system can be found in Figures 6 and 7 and will be explained in detail in the following subsections. Additional examples of how our algorithm works can be found in our technical report [8].

4.1 Weakest precondition

For simplicity, we are presenting our algorithm for single-input-single-output nodes, and so we do not describe the weakest precondition for merge nodes and

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(mustPointTo(Y,X)@in \wedge hasConstValue(Z,C)@in) \vee \\ (mustNotPointTo(Y,X)@in \wedge hasConstValue(X,C)@in) \vee \\ (hasConstValue(Z,C)@in \wedge hasConstValue(X,C)@in)\end{array}}{\begin{array}{c}(\forall\sigma.\big[\sigma(\&X)=\sigma(Y)\big] \wedge \forall\sigma.\big[\sigma(Z)=\sigma(C)\big]) \ \vee (\forall\sigma.\big[\sigma(\&X)\neq\sigma(Y)\big] \wedge \forall\sigma.\big[\sigma(X)=\sigma(C)\big]) \vee \\ (\forall\sigma.\big[\sigma(Z)=\sigma(C)\big] \wedge \forall\sigma.\big[\sigma(X)=\sigma(C)\big])\end{array}}\ fact\ match\ (4)}{\begin{array}{c}\forall\sigma.\big[\sigma(\&X)=\sigma(Y)\wedge\sigma(Z)=\sigma(C)\big] \ \vee \forall\sigma.\big[\sigma(\&X)\neq\sigma(Y)\wedge\sigma(X)=\sigma(C)\big] \ \vee \\ \forall\sigma.\big[\sigma(Z)=\sigma(C)\wedge\sigma(X)=\sigma(C)\big]\end{array}}\ logNorm\ (3)}{\forall\sigma.\big[(\sigma(\&X)=\sigma(Y)\wedge\sigma(Z)=\sigma(C)) \ \vee (\sigma(\&X)\neq\sigma(Y)\wedge\sigma(X)=\sigma(C))\big]}\ \forall\ resolution\ (2)}{\forall\sigma.\big[(\sigma(\&X)=\sigma(\&(*Y))\wedge\sigma(Z)=\sigma(C)) \ \vee (\sigma(\&X)\neq\sigma(\&(*Y))\wedge\sigma(X)=\sigma(C))\big]}\ \&*_E\ (1)$$

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

branch nodes.[7] Computing the weakest precondition for a `skip` statement is simple because it does not modify any values. If s is a `skip`, we therefore have $wp(s,\alpha) = \alpha$. All other statements are treated uniformly as assignments: `decl` $x := e$ assigns x the value of e, $x :=$ `new` assigns x the special token $newloc$,[8] and, finally, $lhs := e$ is a regular assignment.

The standard weakest precondition rule for an assignment, $LHS := E$, substitutes all occurrences of the left hand side for the expression on the right in a predicate α. The presence of Rhodium variables and pointer dereferences in α and LHS make syntactic substitution inadequate. Because Rhodium variables range over IL variables, the result of the substitution needs to depend on whether or not the Rhodium variables in LHS and those in α refer to the same IL variables. Similarly, if either LHS or α contains pointer dereferences, then syntactic replacement will not take into account the fact that LHS may be aliased with terms in α.

In order to solve these two problems, we need to find which parts of α may be modified by the assignment. We use the weakest precondition technique from SLAM in order to do this [1]. Define a location to be either a variable or a variable dereference. Now consider $wp(L := E, \alpha)$, where L is a location. Let L' be some location mentioned in α. There are two possible cases: if L and L' are aliases the assignment modifies L', and L' in α must be replaced with E; if L and L' are not aliases α remains unchanged.

For a predicate α, an expression E and locations L and L', we define $\alpha[L, E, L']$ as follows (where $\sigma(\&L) = \sigma(\&L')$ is used to determine if locations L and L' are aliased):

[7] For such nodes, wp would take input and output edges as additional parameters.
[8] Because all of our rewrites are done on a single statement all references to $newloc$ refer to the same location.

$$\alpha[L, E, L'] \triangleq \begin{array}{l}(\sigma(\&L) = \sigma(\&L') \wedge \alpha[L' \mapsto E]) \vee \\ (\sigma(\&L) \neq \sigma(\&L') \wedge \alpha)\end{array}$$

If $L_1, L_2, \ldots, L_n$ are all the locations in α, then $wp(L := E, \alpha)$ is defined to be $\alpha[L, E, L_1][L, E, L_2] \ldots [L, E, L_n]$.

For constant propagation, the weakest precondition of $\sigma(X) = \sigma(C)$ with respect to a statement of the form $Y := K$ is:

$$(\sigma(\&X) = \sigma(\&Y) \wedge \sigma(K) = \sigma(C)) \vee (\sigma(\&X) \neq \sigma(\&Y) \wedge \sigma(X) = \sigma(C))$$

In our IL, two variables have the same address only if they are syntactically the same and two constants have the same value only if they are syntactically the same constant, so this formula is equivalent to $(X \doteq Y \wedge K \doteq C) \vee (X \not\doteq Y \wedge \sigma(X) = \sigma(C))$, which is exactly the predicate we used in the overview example from section 3.

As another example, the weakest precondition of $\sigma(X) = \sigma(C)$ with respect to a statement of the form $*Y := Z$ is:

$$(\sigma(\&X) = \sigma(\&(*Y)) \wedge \sigma(Z) = \sigma(C)) \vee (\sigma(\&X) \neq \sigma(\&(*Y)) \wedge \sigma(X) = \sigma(C))$$

In our IL, $\sigma(\&(*Y)$ simplifies to $\sigma(Y)$, so the above formula essentially does case analysis based on whether or not Y points to X.

The standard weakest precondition operator *wp* does not include quantification over σ; the quantification is implicit. In our algorithm, it will be useful to make the quantification over σ explicit. We therefore use a version of weakest precondition called $wp_\forall$ which makes this quantification explicit.

We use n to refer to the node that we are currently computing the weakest precondition for. Furthermore, we assume that Σ_{in} refers to the set of all concrete program states that can appear on the incoming CFG edge of n for any execution of the particular IL program being analyzed. We define Σ_{out} similarly. Finally, $wp_\forall$ is defined as follows (where *wp* is the previously defined weakest precondition function without quantification):

$$wp_\forall(s, (\forall \sigma \in \Sigma_{out} \ . \ \alpha)) \triangleq \forall \sigma \in \Sigma_{in} \ . \ wp(s, \alpha)$$

Henceforth, we will use the abbreviation $\forall\sigma.\big[\alpha\big]$ to denote $\forall \sigma \in \Sigma_{in} \ . \ \alpha$.

In this model where quantifiers are explicit, the meaning of $hasConstValue(X, C)@in$ would be $\forall \sigma \in \Sigma_{in} \ . \ (\sigma(X) = \sigma(C))$, which using our abbreviation would be $\forall\sigma.\big[\sigma(X) = \sigma(C)\big]$.

4.2 Normalization

The *Normalize* function transforms a given formula into an equivalent formula that is in a standard normal form. This process involves two kinds of normalization: logical normalization, which is used to put the formula in disjunctive normal form; and IL normalization, which puts terms and expressions in a normal form based on the semantics of our IL. IL normalizations include such simplifications as rewriting $*(\&Y)$ to Y.

Context Rules

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

IL Normalization

Term Rewrite Rules

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

Formula Rewrite Rules

$[\doteq_{I_\&}]\quad \sigma(\&X) = \sigma(\&Y) \rightsquigarrow X \doteq Y$ $\qquad$ $[\not\doteq_{I_\&}]\quad \sigma(\&X) \neq \sigma(\&Y) \rightsquigarrow X \not\doteq Y$

$[\doteq_{I_c}]\quad \sigma(K) = \sigma(C) \rightsquigarrow K \doteq C$ $\qquad$ $[\not\doteq_{I_c}]\quad \sigma(K) \neq \sigma(C) \rightsquigarrow K \not\doteq C$

$[\mathsf{T}_{I_{c\&}}]\quad \sigma(C) \neq \sigma(\&X) \rightsquigarrow \mathit{true}$ $\qquad$ $[\mathsf{F}_{I_{c\&}}]\quad \sigma(C) = \sigma(\&X) \rightsquigarrow \mathit{false}$

$[\mathsf{T}_{I_{cn}}]\quad \sigma(C) \neq \mathit{newloc} \rightsquigarrow \mathit{true}$ $\qquad$ $[\mathsf{F}_{I_{cn}}]\quad \sigma(C) = \mathit{newloc} \rightsquigarrow \mathit{false}$

$[\mathsf{T}_{I_{n\&}}]\quad \mathit{newloc} \neq \sigma(\&Y) \rightsquigarrow \mathit{true}$ $\qquad$ $[\mathsf{F}_{I_{n\&}}]\quad \mathit{newloc} = \sigma(\&Y) \rightsquigarrow \mathit{false}$

$[\mathsf{T}_{I_{\&op}}]\quad \sigma(\&X) \neq \sigma(op\ T_1\ \ldots\ T_n) \rightsquigarrow \mathit{true}$ $\qquad$ $[\mathsf{F}_{I_{\&op}}]\quad \sigma(\&X) = \sigma(op\ T_1\ \ldots\ T_n) \rightsquigarrow \mathit{false}$

$[\mathsf{T}_{I_=}]\quad \sigma(T) = \sigma(T) \rightsquigarrow \mathit{true}$ $\qquad$ $[\mathsf{F}_{I_{\neq}}]\quad \sigma(T) \neq \sigma(T) \rightsquigarrow \mathit{false}$

$[\mathsf{T}_{I_{\doteq}}]\quad T \doteq T \rightsquigarrow \mathit{true}$ $\qquad$ $[\mathsf{F}_{I_{\doteq}}]\quad T \doteq T' \rightsquigarrow \mathit{false}$ (if T and T' are distinct)

$$[op_{expand}]\quad \begin{array}{l} F^t[\sigma(op\ T_1\ \ldots\ T_n)] \rightsquigarrow \\ \quad \exists\, C_1, \ldots, C_n\ .\ \sigma(C_1) = \sigma(T_1) \wedge \ldots \wedge \sigma(C_n) = \sigma(T_n) \wedge F^t[\mathit{eval}(op\ C_1\ \ldots\ C_n)] \\ \hfill \text{(where all variables in } T_i \text{ are free in } F^t) \end{array}$$

Logical Normalization

[*logNorm*] Rules which cause conversion to a standard disjunctive normal form.

Fig. 6. Normalization Rules

We express the normalization process as a rewrite system in Figure 6, where $x_1 \rightsquigarrow x_2$ says that the term or formula x_1 is rewritten to x_2. We use $F^t[\cdot]$ to represent a context F^t with a term hole, $F[\cdot]$ to represent a context F with a formula hole, and $F^+[\cdot]$ to represent a context F^+ with a formula hole that appears in a positive position in F^+.

The *context rules* in Figure 6 allow terms and subformulas to be rewritten inside of a formula. *Logical normalization rules*, the details of which are not shown in Figure 6, are used to put the formula in a logical normal form. Step (3) of Figures 4 and 5 shows a logical normalization step where a universal quantifier is pushed inside of a conjunction.

The *IL normalization rules* in Figure 6 are used to simplify the formula based on the semantics of our IL. Language normalization rules come in two categories: *term rewrite rules* and *formula rewrite rules*. Term rewrite rules are used to simplify terms. For example, step (1) of Figure 5 uses the $*\&$ elimination rule $[*\&_E]$ to simplify $\sigma(*(\&Y))$ to $\sigma(Y)$. Formula rewrite rules are used to simplify formulas. Most of these rules simplify formulas to either *true* or *false* based on properties of the IL. Step (1) of Figure 4 uses three common IL normalizations: the $[\doteq_{I_\&}]$ rule is used to simplify $\sigma(\&X) = \sigma(\&Y)$ to $X \doteq Y$; the $[\not\doteq_{I_\&}]$ rule is used to simplify $\sigma(\&X) \neq \sigma(\&Y)$ to $X \not\doteq Y$; and finally, the $[\doteq_{I_c}]$ rule is used to simplify $\sigma(K) = \sigma(C)$ to $K \doteq C$. Each of these rules allows terms to be compared syntactically when there is a one-to-one correspondence between an AST and a value.

The *Normalize* function applies the normalization rules from Figure 6 until none are applicable anymore. For efficiency, this process can be staged into three steps: first apply IL term rewrite rules, then apply IL formula rewrite rules, and finally put the formula in logical normal form. This staging misses no opportunities for simplification because later stages do not introduce new opportunities for applying rules from earlier stages.

4.3 Mapping Formulas to Facts

The $MapFacts$ function takes a formula ϕ and a set S of fact schema declarations, and tries to replace subformulas of ϕ with facts that are instances of schemas in S. All the schema meanings are first normalized and then the schema declarations in S are sorted in decreasing order of meaning size.[9] The algorithm then proceeds through this list, matching each schema meaning greedily against all possible subformulas. When a match is found, the subformula of ϕ is replaced with the matching fact.

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

One subtlety is that Rhodium dataflow facts are must facts and so the connection between a fact in a formula and its meaning is unidirectional: the presence of a fact on an edge implies that its meaning holds at that edge, but not the other way around. As a result, the absence of a Rhodium fact provides no information whatsoever, and it certainly does *not* imply that the meaning of the missing fact does not hold. Consequently, facts should only be matched in positive positions of the formula ϕ. Consider, for example, the case where $\phi = \neg(\forall\sigma.[\sigma(X) = \sigma(C)])$. It would be unsound to transform this into $\neg\mathit{hasConstValue}(X, C)@\mathit{in}$, because the formula $\neg\mathit{hasConstValue}(X, C)@\mathit{in}$ simply states the absence of $\mathit{hasConstValue}(X, C)$ on the input edge, which does *not* imply $\neg(\forall\sigma.[\sigma(X) = \sigma(C)])$.

4.4 Strengthening

If *Normalize* and *MapFacts* are not able to remove all occurrences of σ from ϕ, then we must strengthen ϕ using the *Strengthen* function. The strengthenings used by *Strengthen* are shown in Figure 7 as rewrite rules. We write $F_1 \rightsquigarrow_S F_2$ to denote that F_1 can be strengthened to F_2, meaning that, from a logical point of view, $F_2 \Rightarrow F_1$.

The simplest kind of strengthening is [$\mathit{strong\ false}_I$], which rewrites a formula to *false*. We use this strengthening when no other strengthenings apply,

[9] Alternatively, if the user has an intuition of which facts would be better to match first, we could allow the user to specify this ordering.

<u>*Context Rules*</u>

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

<u>*Strengthening Rules*</u>

[*strong* F_I]	$F \rightsquigarrow_S \mathit{false}$
[*syntactic*]	$\sigma(T_1) = \sigma(T_2) \rightsquigarrow_S T_1 \doteq T_2$
[$\forall \vee$ *push*]	$\forall x.\big[F_1 \vee F_2\big] \rightsquigarrow_S \forall x.\big[F_1\big] \vee \forall x.\big[F_2\big]$
[$\forall$ *resolution*]	$\forall x.\big[(F_1 \wedge F_2) \vee (\neg F_1 \wedge F_3)\big] \rightsquigarrow_S \forall x.\big[F_1 \wedge F_2\big] \vee \forall x.\big[\neg F_1 \wedge F_3\big] \vee \forall x.\big[F_2 \wedge F_3\big]$

Fig. 7. Strengthening Rules

but there are still references to σ in ϕ. In this case, for each occurrence of σ, we find the smallest enclosing subformula of ϕ that is in a positive position in ϕ, and replace it with *false*.

A less conservative kind of strengthening is to change value equality into syntactic equality, as shown in the [*syntactic*] rule: if two terms are syntactically equal, in other words $T_1 \doteq T_2$, then they evaluate to the same thing, in other words $\sigma(T_1) = \sigma(T_2)$. This is a strengthening because the implication does not hold the other way: $\sigma(T_1) = \sigma(T_2)$ does not necessarily imply $T_1 \doteq T_2$.

A third kind of strengthening is to push universal quantifiers into disjunctions, as shown in [$\forall \vee$ *push*]. Immediately after computing the weakest precondition, the σ quantifier occurs at the outermost level of ϕ. However, individual meanings of fact schemas will match nested subformulas only if the quantifier occurs as part of the subformula, and therefore pushing quantifiers inwards is critical. The normalization process takes care of pushing $\forall$'s into conjunctions (this leads to an equivalent formula). The [$\forall \vee$ *push*] rule is what allows our algorithm to push $\forall$'s into disjunctions.

The *Strengthen* function considers the rules from Figure 7 in the following order: [$\forall \vee$ *push*], then [*syntactic*], and finally [*strong* false_I]. *Strengthen* applies the first rule that matches in this list and it only performs one strengthening per invocation.

If the *Strengthen* function only used [$\forall \vee$ *push*] to push quantifiers through disjunctions, then step (2) of Figure 5 would only generate the first two cases of the disjunction, and the generated rule would be:

$$\begin{gathered}(\mathit{mustPointTo}(Y, X)@\mathit{in} \wedge \mathit{hasConstValue}(Z, C)@\mathit{in}) \vee \\ (\mathit{mustNotPointTo}(Y, X)@\mathit{in} \wedge \mathit{hasConstValue}(X, C)@\mathit{in})\end{gathered}$$

The first disjunct is applicable if we can statically determine $\mathit{mustPointTo}(X, Y)$ and the second disjunct is applicable if we can statically determine $\mathit{mustNotPointTo}(X, Y)$. However, these two rules do not cover a third situation, in which we can statically determine neither $\mathit{mustPointTo}(X, Y)$ nor $\mathit{mustNotPointTo}(X, Y)$. In this third situation, we could still soundly propagate $\mathit{hasConstValue}(X, C)@\mathit{out}$ as long as $\mathit{hasConstValue}(X, C)@\mathit{in} \wedge \mathit{hasConstValue}(Z, C)@\mathit{in}$ holds. We can recover this third case automatically and generate useful rules for it using resolution.

When applied to disjunctive normal form, resolution says that $(A \wedge B) \vee (\neg A \wedge C)$ is equivalent to $(A \wedge B) \vee (\neg A \wedge C) \vee (B \wedge C)$. In our case, we can use resolution to add the missing disjunct. At first, it may seem counterproductive to add disjuncts using resolution, because the added disjuncts are logically redundant. However, the key idea is that these additional disjuncts will reduce the amount of precision that is lost in later $[\forall \vee \mathit{push}]$ strengthenings.

Consider for example $\forall\sigma.\big[(A \wedge B) \vee (\neg A \wedge C)\big]$. Pushing the quantifier through disjunctions and conjunctions produces the stronger formula $(\forall\sigma.\big[A\big] \wedge \forall\sigma.\big[B\big]) \vee (\forall\sigma.\big[\neg A\big] \wedge \forall\sigma.\big[C\big])$. Alternatively, we could apply resolution to the original formula and get $\forall\sigma.\big[(A \wedge B) \vee (\neg A \wedge C) \vee (B \wedge C)\big]$. Although at this point the disjunct $B \wedge C$ is redundant, once we push the quantifier inward, we get $(\forall\sigma.\big[A\big] \wedge \forall\sigma.\big[B\big]) \vee (\forall\sigma.\big[\neg A\big] \wedge \forall\sigma.\big[C\big]) \vee (\forall\sigma.\big[B\big] \wedge \forall\sigma.\big[C\big])$, and in this formula, the disjunct $\forall\sigma.\big[B\big] \wedge \forall\sigma.\big[C\big]$ is *not* redundant. The additional disjunct covers the case in which neither $\forall\sigma.\big[A\big]$ nor $\forall\sigma.\big[\neg A\big]$ can be determined statically. In fact, the formula produced by simply pushing the quantifiers implies the formula produced by doing resolution and then pushing the quantifiers. As a result, the latter strengthening loses less precision, and therefore is a better strengthening to apply.

Because the disjuncts introduced by resolution are redundant and therefore useless until the $[\forall \vee \mathit{push}]$ rule is applied, we have merged $[\forall \vee \mathit{push}]$ with resolution to produce the combined rule $[\forall\ \mathit{resolution}]$, as shown in Figure 7. The *Strengthen* function considers this rule before the other three strengthening rules. The $[\forall\ \mathit{resolution}]$ rule is used in step (2) of the examples in Figures 4 and 5. The use of this resolution rule has uncovered useful cases that we had not thought of previously when writing the flow functions by hand (and some cases that we had written by hand but would have missed without resolution).

5 Current and Future Work

We are currently trying our technique on additional dataflow fact schemas by hand, for example an available expressions fact schema. Once this is done, we will start implementing our algorithm in a prototype tool for inferring Rhodium flow functions. After this prototype is completed, there are several possible directions for future work.

One direction would be to infer not only rules but also new fact schemas. Consider for example the derivation from Figure 5, but this time we will assume that the user has only provided the *hasConstValue* schema, and not the *mustPointTo* and *mustNotPointTo* schemas.

The derivation will be the same as in Figure 5, except that step (4) will lead to the following formula:

$$
\begin{aligned}
&(\forall\sigma.\big[\sigma(\&X) = \sigma(Y)\big] \wedge \mathit{hasConstValue}(Z, C)@\mathit{in}) \vee \\
&(\forall\sigma.\big[\sigma(\&X) \neq \sigma(Y)\big] \wedge \mathit{hasConstValue}(X, C)@\mathit{in}) \vee \\
&(\mathit{hasConstValue}(Z, C)@\mathit{in} \wedge \mathit{hasConstValue}(X, C)@\mathit{in})
\end{aligned}
$$

Because *mustPointTo* and *mustNotPointTo* were not provided as fact schemas, there is no way to map the remaining σ's onto facts, and so our current algorithm will have to resort to strengthenings: it will be forced to use [*strong* false_I] to remove the first two disjuncts, leaving only the third one. Our goal is to extend our algorithm so that instead of strengthening in the above case we would infer the need for two new fact schemas: one with meaning $\sigma(\&X) = \sigma(Y)$, and one with meaning $\sigma(\&X) \neq \sigma(Y)$. These are exactly the *mustPointTo* and *mustNotPointTo* schemas that we previously assumed were given to us. Our algorithm would have to infer rules for these new fact schemas, which may in turn lead to more fact schemas. The key challenge will be to find good heuristics for choosing when to infer new fact schemas and when to strengthen so that this loop ends in practice.

Another direction for future work is to infer a set of fact schemas for justifying a particular optimizing transformation. The user would only provide an optimization goal of the form "try to transform s_1 to s_2", and our system would essentially infer the dataflow analysis to drive this optimization: it would find the fact schemas to justify a Rhodium transformation rule of the form ***if*** $stmt(s_1) \wedge \psi$ ***then transform*** s_2, and then it would find the rules to propagate facts that are instances of these schemas (potentially inferring more fact schemas in the process). In the very long term, we could also try to infer the optimizing transformations themselves, or provide programmers with various tools to help them find useful optimizing transformations.

6 Related Work

Our work is closely related to predicate abstraction [3,2,1]. 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 out all possible abstract transitions. The generated flow functions consist of those transitions that the theorem prover was able to validate. Unlike the predicate-abstraction approach, our technique handles infinite-domain fact schemas such as $hasConstValue(X, C)$, where C ranges over arbitrary constants.

The recent work of Reps, Sagiv, and Yorsh [7,9] 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 (which 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. 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. In contrast, our approach generates flow functions that are specific to the domain specified by the user. Furthermore, their flow function is computationally expensive. Each invocation of their flow function uses an iterative approximation technique that makes successive calls to a decision procedure (a theorem prover). In contrast, our generated flow functions only perform syntactic checks when they are executed.

Finally, our work is also related to the HOIST system for automatically deriving static analyzers for embedded systems [6]. 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.

7 Conclusion

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

References

[1] Ball, T., R. Majumdar, T. Millstein and S. K. Rajamani, *Automatic predicate abstraction of C programs*, in: *Proceedings of ACM SIGPLAN PLDI'2001*, Snowbird, Utah, USA, 2001.

[2] Das, S., D. L. Dill and S. Park, *Experience with predicate abstraction*, in: *Proceedings of the 11th International CAV Conference*, 1999.

[3] Graf, S. and H. Saidi, *Construction of abstract state graphs of infinite systems with PVS*, in: *Proceedings of the 9th International CAV Conference*, 1997.

[4] Lerner, S., T. Millstein and C. Chambers, *Automatically proving the correctness of compiler optimizations*, in: *Proceedings of ACM SIGPLAN PLDI'2003*, San Diego, California, USA, 2003.

[5] Lerner, S., T. Millstein, E. Rice and C. Chambers, *Automated soundness proofs for dataflow analyses and transformations via local rules*, in: *Conference Record of the 32nd ACM SIGPLAN-SIGACT POPL*, Long Beach, California, USA, 2005.

[6] Regehr, J. and A. Reid, *HOIST: A system for automatically deriving static analyzers for embedded systems*, in: *Proceedings of the 11th International ASPLOS Conference*, Boston, Massachusetts, USA, 2004.

[7] Reps, T., M. Sagiv and G. Yorsh, *Symbolic implementation of the best transformer*, in: *Proceedings of VMCAI 2004*, Venice, Italy, 2004.

[8] Rice, E., S. Lerner and C. Chambers, *Automatically inferring sound dataflow functions from dataflow fact schemas*, Technical Report UW-CSE-2005-02-03, University of Washington (2005).

[9] Yorsh, G., T. Reps and M. Sagiv, *Symbolically computing most-precise abstract operations for shape analysis*, in: *Proceedings of TACAS'2004*, Barcelona, Spain, 2004.