Technical Report

Guided Reverse Analysis of Contingent Properties

Jeffery von Ronne, Keyvan Nayyeri, and Zi Yan

Technical Report CS-TR-2011-008
Department of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249

Abstract. Many useful properties that might be discovered from a component in a large software system do not depend solely on the code for that component but rather are contingent upon properties of that component's environment. This paper describes a novel approach for adapting modular analyses, based on abstract interpretation, to run in reverse direction in order to discover a property that—if assumed about the environment of a component—is sufficient to guarantee that some contingent property of that component is also satisfied. This reverse analysis is guided by the results of the original modular analysis, so that it can choose a suitable sufficient condition when there is no best one. Furthermore, the conditions that must be met by a reverse analysis are formally presented, and the process of designing a guided reverse analysis is illustrated through the development of a constant-value analysis.

1 Introduction

Large software systems are notoriously difficult to understand, build, maintain, and verify. These difficulties can be ameliorated by breaking the software system down into smaller components that interact only through well-defined interfaces. To the extent that this is done, the software designers, developers, and testers benefit by being able to correctly reason about each smaller component without needing to consider all of the details about and interactions with the component's larger environment.

Dynamic extensibility is widely developed (e.g., Eclipse plug-ins) in order to easily expand or alter the behavior of a software by adding new components or modifying existing ones without altering or even reinstalling the rest of the components in the system. Dynamic class loading in the Java programming language is a good example of the infrastructure provided for such scenarios where a component can be loaded to the system.

Compiler optimizations, however, depend on the information that can be collected from the code, and the use of components in software, which brings encapsulation to the table, introduces new challenges because performing an analysis on the code of a component also relies on the information gathered from other components that it depends on or is depended on by. It is possible to analyze all such components and get the results that we desire but the challenge is when we want to perform an analysis without any knowledge of the environment of a component or regardless of that, so we can perform the analysis in all cases.

This is very common in dynamic extensibility because components of a codebase may be modified after the original analysis. In this case, we would not be able to determine whether the original analysis is still valid or not, so we would not know whether we could safely continue to use optimizations that rely on those analysis results.

One possible way to work around this issue is to re-analyze the code of a program with each and every change (addition or modification) but this is expensive. The cost is proportional to the analysis cost plus the number of changes in the program, and the runtime system would need to re-perform this analysis dynamically each time a new component is loaded. To solve this problem, we are developing a framework for contingent analysis and optimization.

To optimize a component using this framework, an inter-component static analysis is first applied to some collection of components (e.g., the Eclipse framework plus some subset of available plugins) in order to discover a property describing the possible behaviors of each component under all executions of the programs formed by composing these components. Based on these properties, a compiler optimization identifies a set of weaker properties in each component that it is interested in because they enable certain optimizations. For each *property of interest* of a component, further analysis is then performed to find a *contingent assumption* about the properties of other components that are sufficient to guarantee that the property of interest soundly approximates the possible behaviors of that component. The component's code can then be accompanied by its properties of interests, contingent assumptions, and evidence that the contingent assumptions entail the corresponding properties of interest. These are made available to the execution runtime (e.g., a modified Java Virtual Machine), which can optimize based on the contingent properties. It will only do so, however, after it has verified the evidence for each contingent property against the component's code and its corresponding contingent assumption. Furthermore, each time a new component is dynamically loaded, the contingent assumptions on which prior optimizations were performed will need to be verified against the (contingent) properties of each newly loaded components. If an assumption fails to be verified, then the optimizations performed based on that assumption will need to be rolled back.

This paper develops a foundational methodology for analysis within the contingent optimization framework. This method uses static analysis to infer contingent assumptions to guarantee desired *properties of interest* associated with particular components of the program, and is described within the framework of abstract interpretation [1]. It consists of developing *reverse functions* for programs' abstract semantic transformers. Intuitively, this reverse function needs to map each property to (an appropriate under-approximation of) the pre-image of that property under the abstract semantic transformer. The challenge is in finding an appropriate under-approximation. First, it should be representable in the domain of the abstract analysis. Second, it should be a property that the program actually has (and can be verified to have using the abstract semantics). This can be accomplished by first using abstract interpretation to calculate the program's fixed point semantics, and then using that property can be used to guide reverse abstract interpretation. This guided reverse function can then be used to find a assumed property describing conditions under which each property of interest holds. In this way as long as any changes to the rest of the program preserve the assumed property, and the components associated with the property of interest is unchanged, the property of interest will continue to hold. Furthermore, optimizations relying on that property of interest will continue to be valid.

The contribution of this paper is four-fold. First, it presents guided reverse analysis as a solution to the problem of discovering assumptions about global properties that can be used to verify specific contingent properties. Second, it formalizes the requirements of a guided reverse analysis in relation to modular abstract fixed-point semantic. Third, it demonstrates, that if suitable guided reverse functions are defined can be provided, then it is possible to automatically find conditions sufficient to guarantee that a contingent property of some component continues to hold as other components change. Fourth, it provides insight as to what is required to develop an effective guided reverse analysis. This is accomplished by developing a guided reverse analysis for a modular constant-value analysis of a simple modular imperative language and discussing what is needed to construct a guided reverse analysis utilizing the octagon-domain.

```
component A is
  proc foo() is
    y := B.bar(3, 5, 7)
    return y
  end
end

component B is
  export bar
  proc bar(a, b, c) is
    x := a + b;
    assert (x = 8)
    y := 0
    return y
  end
end
```

Fig. 1. An Example Program

2 Background

Abstract interpretation [1] is an approach to static analysis that soundly approximates the semantics of programs as the fixed point of a monotonic function (transformer) over a domain of abstract properties. In the case of state-based collecting semantics, a property associates each program location with (an upper-approximation of) the set of possible program states that hold at that program location. Instead of explicitly enumerating the members of these sets of states, properties are taken from some abstract domain, whose elements represent concrete sets of states. These properties are ordered according by inclusion of possible state, such that larger elements represent more approximate information about possible executions. The program's code is used to derive a transformer function whose least fixed point is the most precise over-approximation of the program's possible behaviors that can be shown to be sound using that transformer and associated abstract domain. Normally, this semantic property is found by iteration of the transformer or an upper-approximation of this semantic property is found through widening [1].

Thus, given a program, abstract interpretation can be used to discover a semantic property that describes an approximation of the program's possible behaviors during execution. In many cases, this semantic property is sufficient to ensure that certain optimizations that can be performed. Furthermore, a weaker property of interest can be identifed by selecting those components of the semantic property.

In contrast, the focus of this paper is on an approach to address the problem of inferring conditions sufficient to guarantee that some property—that is known to hold—continues to hold as a program evolves. That is, given some *property of interest* associated with some component, the sufficient condition inference problem is how to determine conditions about the rest of the program that are both sound reflections of the current program semantics and are also sufficient to ensure some *property of interest*. That is, the condition sufficiently constrains the evolution of the component's environment, such that if the component is unchanged its property of interest will continue to hold as long as this condition is satisfied by all program executions. As a more concrete example of the concretely, an instance of this problem would be finding a precondition for the procedure `bar` of component `B` of the program in 1 sufficient to establish the correctness of the assertion. In this example, the strongest precondition (that is satisfied by the call in component `A`) is $a = 3 \wedge b = 5 \wedge c = 7$, whereas the weakest precondition matching this criteria is $a + b = 8$.

Generally, it is preferred to identify weaker rather than stronger preconditions, since this will put less constraints on when a property of interest (and any optimization performed assuming the correctness of a property of interest) is known to remain valid. For example, $a + b = 8$ would be preferred over $a = 3 \wedge b = 5 \wedge c = 7$ for the property $x = 8$. If component A is modified so that it calls `B.bar(6,2,7)` or `B.bar(3,5,12)`, then the precondition of $a + b = 8$ would still be satisfied, but the precondition $a = 3 \wedge b = 5 \wedge c = 7$ would not.

```
component A is
  proc foo() is
    y := B.bar(3, 5, 7)
    return y
  end
end
```

```
component B is
  export bar
  proc bar(a, b, c) is
   { forward:                      exact reverse:              guided reverse:             }
   { x = ⊤, a = 3, b = 5, c = 7    x = 8, a + b = 8, c = ⊤     x = ⊤, a = 3, b = 5, c = ⊤  }
    x = a + b;
   { forward:                      exact reverse:              guided reverse:             }
   { x = 8, a = 3, b = 5, c = 7    x = 8, a = ⊤, b = ⊤, c = ⊤  x = 8, a = ⊤, b = ⊤, c = ⊤  }
    assert (x = 8)
    y := 0
    return y
  end
end
```

Fig. 2. Example of a Constant-Value Analysis

For this reason, although the property discovered by forward abstract interpretation as the least fixed point of the transformer can be used to successfully find that $x = 8$, it can only derive the possible parameter values from the calls that actually occur in the program and not how they are used. (See the first column of analysis results in Figure 2.) This could be used as the precondition, but it is over-determined by unrelated facts (e.g., $c = 7$) that happen to be true in the program but do not, in fact, need to be preserved during program evolution. Thus, simply extracting sufficient conditions from the least fixed point property of the abstract interpretation is not a good solution to the sufficient condition problem.

It seems natural, then, to start with the property of interest and reason in reverse to find appropriate sufficient conditions. For the case of properties described by forward, state-based analyses, Dijkstra's weakest-precondition reasoning [2] could be employed, but since we want to automatically optimize and verify properties of programs with loops and recursion, an automated and memory-efficient strategy for finding loop invariants would be needed. For an analysis based on forward, state-based abstract interpretation, a reverse analysis [3, 4]. is roughly equivalent to weakest preconditioning reasoning with automatic discovery of loop invariants. Specifically, a reverse analysis is created by using the same abstract domain and then defining a new abstract transformer as the reverse of the analysis's abstract semantic transformer. For a transformer F, which is $\sqcup$-distributive, a best reverse transformer can be defined using:

$$R(x) = \bigsqcup \{y \mid F(y) \sqsubseteq x\}$$

R maps each property x in F's co-domain to the least-precise property that is a pre-image of x under F; this least-precise property is known to exist, because F is distributive. For a conventional dataflow analysis, applying this reversal to its semantic transformer is equivalent to reversing each transfer function, and reversing the direction of the analysis (e.g., forwards to backwards). When R can be defined in this way, the precondition of a property of interest m would be able to be found by taking greatest fixed point of R at most m, and a property is a sufficient condition if and only if it is at most this fixed point.

```
component A is
  proc bar() is
    y1 := B.foo(0)
    y2 := B.foo(10000)
    y3 := B.foo(90)
    return y3
  end
end
```

```
component B is
  export foo
  proc foo(int i) is
                    { forward:            exact reverse:          guided reverse:    }
                    { $0 \le i \le 10000$   $i \le -5 \lor 0 \le i$    $0 \le i \le \infty$ }
    if (i > -5) then
                    { forward:            exact reverse:          guided reverse:    }
                    { $0 \le i \le 10000$   $0 \le i \le +\infty$      $0 \le i \le \infty$ }
      if (i <= 95) then
                    {                     exact reverse:                             }
                    { forward:            $0 \le i \le 95 \lor$       guided reverse:    }
                    { $0 \le i \le 95$      $101 \le i \le +\infty$    $0 \le i \le 95$    }
        if (i <= 100) then
                    { forward:            exact reverse:          guided reverse:    }
                    { $0 \le i \le 95$      $0 \le i \le 95$          $0 \le i \le 95$    }
          assert (0 ≤ i ≤ 95)
        end
      end
    end
    y := 0
    return y
  end
end
```

Fig. 3. An Interval Range Analysis Example

Unfortunately, not all analyses have distributive transformers. In non-distributive analyses, there may not always be a most-general pre-image of a property under the semantic transformer. An example of this would be the application of interval domain analysis to the program in Figure 3. If this program was analyzed with analysis that associates each variable at each program location with a lower bound and an upper bound taken from $\mathbb{Z} \cup \{-\infty, \infty\}$, the result would be the 'forward' analysis result shown in Figure 3. As can be seen in the 'exact reverse' column, the weakest precondition for the statement `if (i <= 100) ...` requires a disjunction of two different intervals. This would not be representable in an abstract domain that maps each variable at each location to exact interval, and it may not always be feasible to extend abstract domains sufficiently to ensure the weakest precondition is representable. If a program's concrete semantics are distributive, then a distributive abstract semantics can be generated by taking the disjunctive completion of non-distributive abstract semantics. The cost of the analysis with this domain will, however, increase exponentially compared to analysis with the original domain. [5]. Thus for efficiency, we reject solutions that do not operate on the original abstract domain.

What, then, is required is a method for finding suitable sufficient preconditions for properties in the abstract domains of non-distributive abstract semantics. This is the role of guided reverse analysis.

3 Overview of Guided Reverse Analysis

The fundamental problem with developing reverse functions for non-distributive abstract semantics is that, although—for efficiency—we want a reverse function that yields a single element in the abstract domain, the least upper-bound of the pre-image of a property under the abstract transformer is not necessarily a member of the pre-image. This would be the case for trying to define reverse transfer functions for an analysis of the program in Figure 3 with an abstract domain where each variable is associated with a single interval. The weakest precondition for the statement `if (i <= 100) ...` , which uses a disjunction, cannot be represented in that abstract domain, so another approach must be taken.

For purposes of finding a sufficient condition, though, under-approximation is sound, so one could use a reverse function that maps each abstract property m to some arbitrary abstract property l that is at most a member of m's pre-image. In our example, this would mean that defining the reverse function associated with the statement `if (i <= 100)`, either so that it maps $0 \leq i \leq 95$ to $0 \leq i \leq 95$ or so that it maps $0 \leq i \leq 95$ to $101 \leq i \leq +\infty$, would produce valid preconditions for the statement.

One problem with this approach, however, is that the sufficient condition produced by the reverse function might not be one that the program actually has. For example, though $i <= -5$ is a valid precondition for the procedure `foo` in Figure 3, it is not actually satisfied by the calls that occur in the program. Therefore, we also wish to require, that whenever the property of interest is discoverable through abstract interpretation of a program using some abstract semantics, that the condition derived by a reverse analysis related to those abstract semantics will be a property that is sound according to the abstract semantics.

Consequently, we require guided reverse function to choose from among the elements of the pre-image of the input property under the abstract semantics, such that the chosen element is at least the the program's abstract fixed point semantics. In order to make it possible to define such functions, we allow the guided reverse functions to be parameterized by a *guidance property* that approximates the programs abstract fixed point semantics. This allows the reverse function to 'peak' at the guidance property in order to decide which of the pre-image elements to choose. Thus, for our interval analysis example, a guided reverse function could look at the guidance of $0 \leq i \leq 10000$ in choosing $0 \leq i$ over $i \leq -5$ and could look at the guidance of $0 \leq i \leq 95$ in choosing $0 \leq i \leq 95$ over $101 \leq i \leq +\infty$.

4 Example: A Constant Value Analysis

4.1 A Program

We now describe a guided reverse analysis that approximates program properties by associating variables at program locations with constant values. The results of applying such analysis to an example program is shown in Figure 2

This analysis assigns a property to each variable at each program location of either an integer value—if the variable is known to have that value—or $\top$—if it is not known that the variable has a constant integer value at that location. An inter-component static analysis could discover that `x` has a constant value of 8, by discovering that `B.bar()` is always called when `a` has a value of 3 and `b` has a value of 5. But there are other possible global assumptions sufficient to guarantee the property of interest (i.e., that `x` is 8). (For example, a property specifying that `a` is 0 and `b` is 8 would also be sufficient.) Unfortunately, in the domain described above, the least upper bound of the sufficient properties is $\top$ and is not itself a sufficient property.

Thus, it is necessary to settle for a property that is sufficient to guarantee the property of interest but is not necessarily the weakest of all possible sufficient properties. Since it is not the weakest, however, care must be taken to find a sufficient property that is weaker than the most precise property discoverable by the whole program analysis. Such a property can be discovered by utilizing the results of the original analysis to guide the reverse analysis.

4.2 A Simple Modular Imperative Language (SMIL)

We will now define a simple modular imperative language (SMIL) that we will then define a constant value analysis for. SMIL is a standard imperative "while" language to which we have added components and procedure calls (cf. [6]). We embellish SMIL programs with some labels to make the discussion easier and assume that distinct elementary blocks are initially assigned distinct labels.

We use the following syntactic categories:

$$a \in \textbf{AExp};\ b \in \textbf{BExp};\ S \in \textbf{Stmt}$$

$$x, y \in \mathbf{Var};\ n \in \mathbf{Num};\ \ell \in \mathbf{Lab};\ p \in \mathbf{Proc};\ c \in \mathbf{Comp}$$

$$op_a \in \mathbf{Op}_a;\ op_b \in \mathbf{Op}_b;\ op_r \in \mathbf{Op}_r$$

The abstract syntax of the SMIL language is given below:

a ::= $x \mid n \mid x_1\ op_a\ x_2$

b ::= true | false | not $b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2$

S ::= $[x := a]^\ell \mid [\text{skip}]^\ell \mid S_1; S_2 \mid$ if $[b]^\ell$ then S_1 end | while $[b]^\ell$ do S |
call $p(x_1, \ldots, x_n)^{\ell_c,\ell_{cn}}_{\ell_{cr},\ell_{xr},\ell_r}$ | call $c.p(x_1, \ldots, x_n)^{\ell_c,\ell_{cn}}_{\ell_{cr},\ell_{xr},\ell_r}$

C ::= component c is $D\ S$ end | $C\ C$

D ::= proc $p(x_1, \ldots, x_n)$ is$^{\ell_n}$ S *return* y^{ℓ_x} end | import $c.p$ | export $p^{\ell_{e_1}}_{\ell_{e_2}}$ | $DD \mid \epsilon$

A program P in the SMIL language consists of a sequence of C declarations for components each of which consists of a sequence of D procedure declarations that include statements. We assume that each program P has a main component with a main procedure whose entry point is the starting point for the program. Each component C can define procedures, import procedures from other components, and export procedures to other components. SMIL follows a call by value strategy for parameter passing and has a single return value for each procedure.

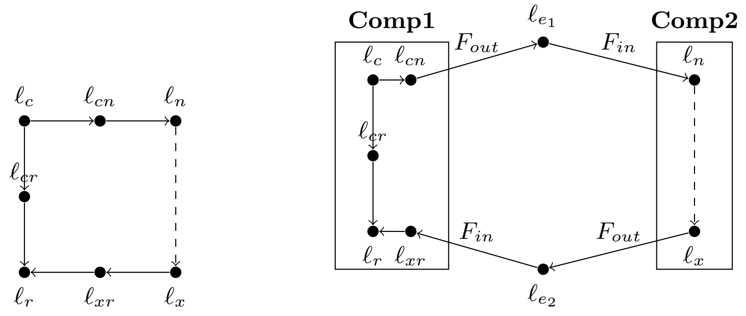

Fig. 4. Intra-component flow and inter-component flow: *each solid arrow represents a control flow; each dashed arrow represents one or more control flows.*

SMIL Control Flow Graph The control flow graph nodes and edges associated with the labeled blocks within the various statements are straightforward, except in the case of procedure calls, which are illustrated in Figure 4. The flow graph nodes and edges contain associated with intra-component procedure calls provide two paths for each call: one connects the entry of the call site ℓ_c to the callee's entry node ℓ_n, through the callee's body to the callee's exit node ℓ_x, and then back to the call-sites return node ℓ_r. This path is used to propagate properties related to parameters and return values. A second path bypasses the callee and goes from the call site ℓ_c to the call site return ℓ_r. This path is used to propagate properties about the local variables of the caller (which will never be modified by the callee). Because our analysis will associate transfer functions with nodes rather than edges, and since different information is propagated on these paths, additional nodes $\ell_{cn}, \ell_{cr}, \ell_{xr}$ are inserted between ℓ_c and ℓ_n, ℓ_c and ℓ_r, and ell_x and ℓ_r, respectively, so that different transfer functions can be applied along those different paths.

The flow graph also differentiates between procedure calls that are between two procedures within a single component versus and procedure calls that cross component boundaries. Additional 'export' nodes being introduced on inter-component calls and returns. This allows us to define properties associated with the 'export' nodes as global program properties and define all other properties as local to particular components.

4.3 The Constant-Value Analysis

The constant value analysis associates each program location with an element of a subset of the following abstract domain with the natural ordering:

$$L = (\mathbf{Var} \cup \{1, \dots, n\}) \rightarrow (\mathbb{Z} \cup \{\bot, \top\}) \tag{1}$$

where **Var** is all the variable names occurring in the program and n is the maximum number of parameters any procedure takes in the program. Note that this domain is finite, so all fixed point iterations will eventually stabilize.

Definitions for the transfer functions associated with different types of nodes can be found in the left column of Table 1. Except for the transfer functions associated with procedure calls, these definitions are straightforward.

Since variable names are local to each procedure invocation, $f_{\ell_{cn}}$ maps functions whose domain is local variables in the caller into function whose domain is positive integers representing parameter positions. The resulting function associates each position values with the same abstract values as the corresponding local variable in the input function. Similarly, f_{ℓ_n} propagates values from position numerals to formal parameter names. f_{ℓ_x} creates a function mapping the number 1 to the procedure's abstract return value. $f_{\ell_{xr}}$ extracts that abstract return value and associates the left-hand-side of the procedure-call assignment with that abstract value.

4.4 Guided Reverse Transfer Function

In order to define a guided reverse analysis for our constant-value analysis, we will define a *guided reverse function (GRF)* for each transfer function that defines the constant-value analysis. Intuitively, a GRF for some function F is a function related to F, such that for a value, it returns an under-approximation of that value's pre-image under F.

Definition 1. *For any complete lattice L_1 and any complete domain L_2, a valid guidance $l \in L_1$ for a target value $m \in L_2$ with respect to a monotonic function $F : L_1 \rightarrow L_2$ is one such that $F(l) \sqsubseteq m$.*

Table 1. Constant Propagation Analysis

$[x := n]^{\ell}$	
$f_{\ell}(l)(v) = \begin{cases} n & \text{if } v = x \\ l(v) & \text{otherwise} \end{cases}$	$r_{\ell}(l)(m)(v) = m(v) \;\; \text{if} \;\; v \neq x_1$
$[x_1 := x_2]^{\ell}$	
$f_{\ell}(l)(v) = \begin{cases} l(x_2) & \text{if } v = x_1 \\ l(v) & \text{otherwise} \end{cases}$	$r_{\ell}(l)(m)(v) = \begin{cases} m(v) & \text{if } v \neq x_1 \wedge v \neq x_2 \\ \top & \text{if } v = x_1 \wedge v \neq x_2 \\ m(x_1) & \text{if } m(x_1) \sqsubseteq m(x_2) \wedge v = x_2 \\ m(x_2) & \text{if } m(x_2) \sqsubset m(x_1) \wedge v = x_2 \end{cases}$
$[x_1 := x_2 \; op \; x_3]^{\ell}$	
$f_{\ell}(l)(v) = \begin{cases} l(x_2) \; op \; l(x_3) & \text{if } v = x_1 \\ l(v) & \text{otherwise} \end{cases}$	$r_{\ell}(l)(m)(v) = \begin{cases} \top & \text{if } v = x_1 \wedge v \neq x_2 \wedge v \neq x_3 \\ l(x_2) & \text{if } v = x_2 \wedge x_3 \neq \top \\ l(x_3) & \text{if } v = x_3 \wedge x_3 \neq \top \\ \top & \text{if } (v = x_2 \vee v = x_3) \wedge x_3 = \top \\ m(v) & \text{otherwise} \end{cases}$
$[skip]^{\ell}$	
$f_{\ell}(l)(v) = l(v)$	$r_{\ell}(l)(m)(v) = m(v)$
$[b]^{\ell}$	
$f_{\ell}(l)(v) = l(v)$	$r_{\ell}(l)(m)(v) = m(v)$
$export \; p^{\ell_{e_1}}_{\ell_{e_2}}$	
$f_{\ell_{e_1}}(l)(v) = l(v)$ $f_{\ell_{e_2}}(l)(v) = l(v)$	$r_{\ell_{e_1}}(l)(m)(v) = m(v)$ $r_{\ell_{e_2}}(l)(m)(v) = m(v)$
$y_1 := call \; p(x_1, \ldots, x_n)^{\ell_c, \ell_{cn}}_{\ell_{cr}, \ell_{xr}, \ell_r}$	
$f_{\ell_c}(l)(v) = l(v)$	$r_{\ell_c}(l)(m)(v) = m(v)$
$f_{\ell_{cr}}(l)(v) = \begin{cases} \bot & \text{if } v = y_1 \\ l(v) & \text{otherwise} \end{cases}$	$r_{\ell_{cr}}(l)(m)(v) = \begin{cases} \top & \text{if } v = y_1 \\ m(v) & \text{otherwise} \end{cases}$
$f_{\ell_{cn}}(l)(i) = \begin{cases} l(x_i) & \text{if } 1 \leq i \leq n \\ \top & \text{otherwise} \end{cases}$	$r_{\ell_{cn}}(l)(m)(v) = \begin{cases} m(i) & \text{if } v = x_i \;\wedge 1 \leq i \leq n \\ \top & \text{if } \forall i. \; v \neq x_i \end{cases}$
$f_{\ell_{xr}}(l)(v) = \begin{cases} l(1) & \text{if } v = y_1 \\ \bot & \text{otherwise} \end{cases}$	$r_{\ell_{xr}}(l)(m)(v) = \begin{cases} m(y_1) & \text{if } v = 1 \\ \top & \text{otherwise} \end{cases}$
$f_{\ell_r}(l)(v) = l(v)$	$r_{\ell_r}(l)(m)(v) = m(v)$
$proc \; p(p_1, \ldots, p_n) \; is^{\ell_n} \; \mathrm{S} \; return \; y_2^{\ell_x} end$	
$f_{\ell_n}(l)(v) = \begin{cases} l(i) & \text{if } v = p_i \;\wedge 1 \leq i \leq n \\ \top & \text{otherwise} \end{cases}$	$r_{\ell_n}(l)(m)(i) = \begin{cases} m(p_i) & \text{if } 1 \leq i \leq n \\ \top & \text{otherwise} \end{cases}$
$f_{\ell_x}(l)(v) = \begin{cases} l(y_2) & \text{if } v = 1 \\ \top & \text{otherwise} \end{cases}$	$r_{\ell_x}(l)(m)(v) = \begin{cases} m(1) & \text{if } v = y_2 \\ \top & \text{otherwise} \end{cases}$

$$\begin{tikzcd}
L_1 & L_2 \\
 & m \arrow[dl, bend right, "B(l)"'] \arrow[dd, phantom, "\sqsupseteq" rotate=90] \\
R(l)(m) \arrow[dr, bend right, "F"'] \arrow[dd, phantom, "\sqsupseteq" rotate=90] & \\
 & F(R(l)(m)) \arrow[dd, phantom, "\sqsupseteq" rotate=90] \\
l \arrow[dr, bend right, "F"'] & \\
 & F(l)
\end{tikzcd}$$

Fig. 5. Guided Reverse Function

Definition 2. *Given a function $F : L_1 \to L_2$ that is a monotonic function from the complete lattice L_1 to the complete lattice L_2, the function $R : L_1 \to L_2 \to L_1$ is a* guided reverse function (GRF) *for F if, given a value $l \in L_1$ and a value $m \in L_2$, where l is a valid guidance for m with respect to F, the following conditions will be satisfied:*

1. *$R(l)$ is a monotonic function*
2. *$F(R(l)(m)) \sqsubseteq m$*
3. *$R(l)(m) \sqsupseteq l$*

These requirements for a GRF are illustrated in Figure 5. We note that for a distributive function F, the non-guided reverse function for distributive function $R(x) = \bigsqcup\{y \mid F(y) \sqsubseteq x\}$ is a GRF for F. In fact it is the best GRF for F and should be whenever it is efficiently computable.

GRF for Constant-Value Analysis of SMIL programs The guided reverse transfer functions for the constant-value analysis are found on the right column of Table 1. For most of the cases, the reverse function produce are straightforward and produce exact pre-images for each property under the corresponding transfer function. Two of the guided reverse functions are particularly noteworthy.

The only guided reverse transfer function definition that uses the guidance l is the one for $[x_1 := x_2 \text{ op } x_3]^\ell$. This is the transfer function definition that would apply for the statement `x = a + b` in the program of Figure 2. Since the result of a binary operation is a constant only when the operands are constants, the guidance must associate constant values with each of the operands. Furthermore the operands must be associated exactly with these values in any property that is both an upper-approximation of the guidance and a pre-image of a property associating the result with a constant. So for these variables in this case, the abstract values from the guidance can be used as the result of the guided reverse transfer function.

The reverse function for assignments of the form $[x_1 := x_2]^\ell$ is also interesting. It does not define $r_\ell(l)(m)(v)$ for $v = x_2$ when no order is defined between $m(x_1)$ and $m(x_2)$. This is acceptable, because the requirements on GRF are conditioned on the guidance justifying the input of the function (i.e.,$f_\ell(l) \sqsubseteq m$).

This condition cannot be met unless $m(x_1) = m(x_2)$ or one of $m(x_1)$ or $m(x_2)$ is either $\top$ or $\bot$.

In the next section, we will develop the guided backwards analysis framework into which these guided reverse functions can be inserted to obtain a complete analysis.

5 Guided Reverse Analysis Framework

5.1 Modular Analysis Setting

Since the purpose of guided reverse analysis is to determine assumptions about a component's environment based on a property of interest for that component, it is necessary for us to formalize the idea of a component's properties and what aspects of the component's environment may affect it.

We assume that a program's *abstract semantics* is given as a *fixed-point semantic specification* [7], where the *modular abstract semantic domain* and the *modular abstract semantic transformer* are defined in terms of fixed-point semantic specifications of each of the program's components and the program's global name space (cf. [8]).

Definition 3. *Each program component $i \in 1 \ldots n$ is assumed to be associated a complete lattice $(L_i, \sqsubseteq_i, \bot_i, \top_i, \sqcup_i, \sqcap_i)$, which serves as that component's abstract semantic domain.*[1] *An additional complete lattice $(L_0, \sqsubseteq_0, \bot_0, \top_0, \sqcup_0)$ is used as an abstract domain for describing global program properties.*

Each component i is also assumed to be associated with an environment-parameterized abstract transform function $F_i : L_0 \to L_i \to L_i$, which is a monotonic function that maps global properties in the domain L_0 into total monotonic functions from L_i to L_i. The component's abstract fixed-point semantics *under an environmental assumption $a \in L_0$ is given by the least fixed point of $F_i(a)$, that is $lfp^{\sqsubseteq_i} F_i(a)$.*

The program's abstract semantic domain is the complete lattice $(L, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ formed by taking the Cartesian product of the abstract domain for global properties with the abstract semantic domains of each of the components ($L = L_0 \times L_1 \times \ldots \times L_n$) under the standard Cartesian order.

The program's abstract semantic transformer $F : L \to L$ is the total monotonic function given by

$$F(x_0, x_1, \ldots, x_n) = (F_0(x1, \ldots, x_n), F_1(x_0)(x_1), \ldots, F_n(x_0)(x_n))$$

where $F_0 : L_1 \times \ldots \times L_n \to L_0$ is the program's global extractor that maps component property tuples to global properties.

In this paper, we will not be concerned with the details of a program or component's concrete semantics. We will instead assume that programs' and their

[1] We will, however, feel free to use $\sqsubseteq$ for $\sqsubseteq_i$, $\bot$ for $\bot_i$, etc. where the i can be understood from the context.

components' abstract fixed point semantics are correct approximations of some concrete semantics. That is, a program's abstract semantic domain, the domain for the program's abstract global properties and the program's components' abstract semantic domains are related to the semantic domains of their concrete fixed point semantics by Galois connections: $(L^\flat, \sqsubseteq^\flat) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (L, \sqsubseteq)$, $(L_0^\flat, \sqsubseteq_0^\flat) \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} (L_0, \sqsubseteq_0)$, $(L_1^\flat, \sqsubseteq_1^\flat) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (L_1, \sqsubseteq_1)$, $\ldots$, $(L_n^\flat, \sqsubseteq_n^\flat) \underset{\alpha_n}{\overset{\gamma_n}{\leftrightarrows}} (L_n, \sqsubseteq_n)$, and their abstract semantic transformers are related to their concrete transformers as follows: $\forall x \in L : \alpha(F^\flat(\gamma(x))) \sqsubseteq F(x)$, $\forall a^\flat \in L_0^\flat, x \in L_1 : \alpha(F(a)_1^\flat(\gamma(x))) \sqsubseteq F_1(\alpha_0(a))(x)$, $\ldots \forall a^\flat \in L_0^\flat, x \in L_n : \alpha(F(a)_n^\flat(\gamma(x))) \sqsubseteq F_n(\alpha_0(a))(x)$. This ensures that $\alpha(\mathrm{lfp}^{\sqsubseteq^\flat}(F^\flat)) \sqsubseteq \mathrm{lfp}^{\sqsubseteq} F$ and $\forall a^\flat \in L_0^\flat : \alpha_1(\mathrm{lfp}^{\sqsubseteq_1^\flat}(F_1^\flat(a))) \sqsubseteq \mathrm{lfp}^{\sqsubseteq_1} F_1(\alpha_0(a))$, $\ldots$, $\forall a^\flat in L_0^\flat : \alpha_n(\mathrm{lfp}^{\sqsubseteq_n^\flat}(F_n^\flat(a))) \sqsubseteq \mathrm{lfp}^{\sqsubseteq_n} F_n(\alpha_0(a))$. Thus, any abstract property that is at least a fixed point of the corresponding abstract transformers can be taken to soundly circumscribe possible the program behavior.

In order to simplify our notation, we assume that the number, position, and abstract domains of components do not change in related programs. We also make the simplifying assumption that all programs, in which a particular component might be embedded, share the same abstract domain for their global properties. This allows us to use the same function F_i to represent the abstract semantics of program component i even if its environment changes. This assumption could be removed, at the cost of increasing the complexity of our notation, by introducing explicit functions for importing the subset of global information that affects each component.

Example 4. When applying the constant-value analysis described in Section 4 to the program of Figure 1, the program's semantic domain is the product of three sub-domains. The first, is an abstract domain for global properties that maps two labels associated with (the call and return through) the B.bar export to functions that associate numerals to abstract values. The second is a component A's abstract domain, which contains properties that map labels associated with declaration of `proc foo` and labels associated with the call to `B.bar` to maps associating variables names and position numbers to abstract values. The third is component B's abstract domain. It maps program labels associated with the declaration of `proc bar` and the assignment statement to functions that map names and position numbers to abstract values.

That the call to B.bar in proc foo of component A passes the parameters $3, 5$, and 7 is a property in component A's abstract domain. That $x = 8$ at the end of `B.bar` is a property in component B's abstract domain. That $B.bar$ is only called (from components other than B) with the parameters $3, 5$, and 7 is a global properties. A more approximate global property would be that $B.bar$ is only called with the parameters approximated by the abstract values $3, 5$, and $\top$). Similarly, that the procedure `bar` returns 0 can be expressed both as a property of component B and as a global property.

The abstract semantic transformers $F_{\texttt{foo}}$ associated with the component foo and $F_{\texttt{bar}}$ associated with component bar are parameterized by the global property, allowing values associated with flows from the pair of nodes associated

B.bar export to be used in those semantic transformers. The global extractor F_0 propagates the information $3, 5$, and 7 are parameters passed to B.bar from component A's local property to the corresponding global property. If there were any other calls to B.bar in the program (from components other than B), the global extract function F_0 would be defined so as to use the join of these for the abstract values associated with the B.bar export.

In this way, the status of the property that $x = 8$ at the end of `B.bar` depends only on component B and the program's global property. If neither of these changes, there will be no change in whether the program's modular abstract semantics will find that $x = 8$.

5.2 Finding Assumptions and Contracts

Having, thus, defined what we mean by modular analysis, let us review the purpose of our guided reverse analysis in light of this definition. Supposing that $(l_0, l_1, \ldots, l_n) \sqsupseteq F(l_0, l_1, \ldots, l_n)$ and that we wish to guarantee some property of interest $m_i \sqsupseteq l_i$, we would like to be able to find an assumption a about the global property such the property of interest is guaranteed to hold $m_i \sqsupseteq \text{lfp}(F_i(a))$. We would also like to find an assume-guarantee pair $(a_j, g_j) \in L_0 \times L_j$ for each component j, such that:

$$\begin{cases} m_i \sqsupseteq g_i \\ a_j \sqsupseteq F_0(g_1, \ldots, g_n) & \text{for each } j \in 1 \ldots, n \\ g_j \sqsupseteq \text{lfp}(F_j(a_j)) & \text{for each } j \in 1 \ldots, n \end{cases}$$

We will also refer to a pair consisting of an assumed global property and a guaranteed local property as an assumption-guarantee contract. Furthermore, we would ideally like a/a_j, to be as weak (great) as possible, but may be able to settle for stronger (lesser) assumptions, as long as they are still at least as weak as the actual property l_0 of the program.

Example 5. As an example, we will refer again to the constant-value analysis described in Section 4 and the program of Figure 1. Consider the property of interest that $x = 8$ at end of `B.bar`, this is a local property of component B. One valid global assumption that is sufficient to guarantee this property is the global property that the first two parameters in all inter-component calls to B.bar() are always 3 and 5. Another valid global assumption that is sufficient to guarantee this property is that the parameters in all inter-component calls to `B.bar` are 3, 5, and 7. The former is the more useful assumption, because it is weaker while still being sufficient to guarantee the property of interest. Thus we can consider the component B to satisfy the contract that it is guaranteed to have the property $x = 8$ at the end of the procedure under the assumption of the global property that all calls to `B.bar` use the parameters 3 and 5.

Guided Reverse Abstract Semantic Transformers This can be accomplished by using guided reverse abstract semantic transformers (GRAST). The requirements of an GRAST are defined in relation to some abstract semantic transformer. The idea is that a GRAST is a function that, under the right circumstances, returns an under-approximation of the pre-image of some property of interest under the abstract semantic transformer. A GRAST, however, is only required to be defined on properties of at least some specified 'guidance property' where the semantic transformer is reductive. The guidance property can be found by iterating over the semantic transformer or by applying widening/-narrowing. The requirements on GRAST ensure that the iterates of a GRAST from a suitable property of interest is required to form a descending chain where each element is at least the guidance property.

Definition 6. *A valid guidance property for a target property m with respect to a an abstract semantic transformer $F : L \to L$ is one such that $F(l) \sqsubseteq l \sqsubseteq m$.*

We note that a valid guidance property l for m with respect to an abstract semantic transformer F exists if and only if the least fixed point of F is a valid guidance property for m with respect to the abstract semantic transformer F.

Definition 7. *Given an abstract semantic transformer $F : L \to L$ that is a monotonic function over an arbitrary complete lattice L, the function $R : L \to L \to L$ is a* guided reverse abstract semantic transformer (GRAST) *for F if, for any property $l \in L$ and a property $m \in L$ such that l is a valid guidance property for m with respect to F, the following conditions will be satisfied:*

1. *$R(l)$ is a monotonic function*
2. *$F(R(l)(m)) \sqsubseteq m$*
3. *$R(l)(m) \sqsupseteq l$*
4. *$R(l)(m) \sqsubseteq m$*

A GRAST R is usually used to calculate $\bigsqcap_k((R(l))^k(m))$ for a property of interest m with a guidance property of l. This is the greatest fixed point of $R(l)$ at most m, which will be instrumental in finding suitable assumptions for desired properties of interest that can be used with the corresponding base analysis F . Note that the following order will exist among these properties:

$$F(l_1) \sqsubseteq l \sqsubseteq \bigsqcap_k((R(l))^k(m)) \sqsubseteq (R(l))^k(m) \sqsubseteq m$$

Lemma 8. *Given any function R that is an GRAST for some abstract semantic transformer F over a domain L and any two properties l and m in L such that $F(l) \sqsubseteq l \sqsubseteq m$, if the sequence $((R(l))^k(m))_k$ stabilizes, the sequence will stabilize at a property $\bigsqcap_k((R(l))^k(m))$ of at least l, of at most m, and at which F is reductive.*[2]

[2] We use the notiation $(t(k))_k$ to denote the infinite sequence $t(0), t(1), \ldots$ where $t(k)$ stands for an arbitrary term with the free variable of k.

Assuming that we have an GRAST suitably related to a component's abstract semantic function we can use them to construct assumptions for properties of interest within that component. We can also use an GRAST for the entire program's abstract semantic function to find complete sets of assumption-guarantee contracts that will be consistent (i.e., all assumptions will be guaranteed) and will guarantee a particular properties of interest. An example of how such GRAST can be constructed will be discussed in Section 4, but first, in the next sections, we will show how such GRAST's can be used to produce assumptions and assumption-guarantee contracts.

Finding A Sufficient Assumption Recall that modular abstract semantic transformers are defined to have the form:

$$F(x_0, x_1, \ldots, x_n) = (F_0(x1, \ldots, x_n), F_1(x_0)(x_1), \ldots, F_n(x_0)(x_n))$$

where each F_i is the abstract semantic transformer of component i. In order to compute an assumption a_0 about the global property that guarantees some property of interest m_i in component i, we require the definition of a function R_i^2 $R_i^2 : L_0 \times L_i \to L_0 \times L_i \to L_0 \times Li$ that is a GRAST for:

$$\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$$

Theorem 9. *If R_i^2 is a GRAST for $\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$ where F_i is the abstract semantic function of some component and $(l_i) \sqsupseteq F_i(l_0)(l_i)$, then the assumption a found by computing:*[3]

$$a = \pi_1 \left(\prod_k (R_i^2(l_0, l_i))^k(\top, m_i) \right)$$

for some property of interest $m_i \sqsupseteq l_i$ is sufficiently strong to guarantee that, for any global property $x_0 \sqsubseteq a$:

$$m_i \sqsupseteq \mathrm{lfp}(F_i(x_0))$$

whenever the sequence $((R_i^2(l_0, l_i))^k(\top, m_i))_k$ converges.

Consequently, any related program, whose component i is unchanged, whose other components may have been changed, and whose global property is stronger than a, will have a property for component i that is stronger than the property of interest m_i.

Modular Verification with Inferred Contracts The ability to find sufficient conditions for a desired property does not in itself allow the property to be verified modularly. Using an GRAST for the program's abstract semantic

[3] We use the notation $\pi_n(e)$, where e is an expression evaluating to a tuple value, to denote the nth element (starting at 1) of that tuple.

transformer, it is possible to construct an assume-guarantee contract for each component, such that all of the guarantees entail both some property of interest and each component's assumptions. Thus the property of interest can be verified by merely checking, for each component, that under that contract's assumptions, the contract's guarantee is a safe approximation of that component's semantics.

The first step of finding such contracts is to use an GRAST for the global program function to find a *globally sufficient assumption*.

Definition 10. *A* globally sufficient assumption and guarantees *for some property of interest m of a program described by a modular abstract semantic function F is the $(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$ defined by:*

$$(a_{0,0}, g_{0,1}, \ldots, g_{0,n}) = \bigsqcap_k R(l_0, l_1, \ldots, l_n)^k (m_0, m_1, \ldots, m_n)$$

where R is a GRAST for F and a guidance $l \sqsubseteq m$ at which $F(l) \sqsubseteq l$.

The *globally sufficient assumption and guarantees* can be calculated as using the sequence $(R(l_0, l_1, \ldots, l_n)^k (m_0, m_1, \ldots, m_n))_k$ whenever that sequence stabilizes. This will be the case if the lattice associated with L is of finite height. The globally sufficient assumptions can then be used with a GRAST related to each component's abstract semantic transformer to find a set of consistent local contracts as follows.

Definition 11. *A* set of consistent local contracts *for some property m with its globally sufficient assumption and guarantees $(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$ of a program described by a modular abstract semantic transformer F (with related $F_0, F_1, \ldots, F_n$) is the assumption-guarantee contracts $(a_{i,0}, g_{i,i}) \ldots (a_{n,0}, g_{n,n})$ for $i \in 1 \ldots n$ defined by:*

$$(a_{i,0}, g_{i,i}) = \bigsqcap_k ((R_i^2(l_0, l_i))^k (m_0, g_{0,i}))$$

where R_i^2 is a GRAST for $\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$ a guidance $l \sqsubseteq m$ at which $F(l) \sqsubseteq l$.

Whenever the sequence $((R_i^2(l_0, l_i))^k (m_0, g_{0,i}))_k$ stabilizes for some $i \in 1 \ldots n$, it can be used to calculate the assumption-guarantee contracts for component i. This will be the case whenever the lattice associated with $L_0 \times L_i$ is of finite height.

Theorem 12. *If $(a_{1,0}, g_{1,1}) \ldots (a_{n,0}, g_{n,n})$ is a set of consistent local contracts for a property of interest $(m_0, m_1, \ldots, m_n)$ of a program described by a modular abstract semantic transformer F, then:*

1. *the local guarantee for each component i holds if the corresponding local assumption hold:*

$$g_{i,i} \sqsupseteq \mathrm{lfp}(F_i(a_{i,0}))$$

2. *the guarantees are sufficient to establish each of the assumptions:*

$$a_{i,0} \sqsupseteq F_0(g_{1,1}, \ldots g_{n,n}) \text{ for } i \in 1 \ldots n$$

3. *the property of interest holds if all of the local assumptions hold (and hence all of the guarantees hold)*

$$m_0 \sqsupseteq F_0(g_{1,1}, \ldots g_{n,n})$$
$$m_i \sqsupseteq g_{i,i}$$

5.3 Standard Construction of Guided Reverse Semantics

We will now, explain how to construct a function meeting the requirements for a GRAST given a programs semantic transformers given a GRF for each of its component's semantic transformer.

Definition 13. *The standard reverse construction for a modular abstract semantic transformer F is a function R defined as:*

$$\begin{aligned}R(l_0, l_1, \ldots l_n)(x_0, x_1, \ldots x_n) = \\ &(\prod_{i=1}^{n}(\pi_1(R_i(l_0, l_i)(x_i))) \sqcap x_0, \\ &\pi_2(R_1(l_0, l_1)(x_1)) \sqcap \pi_1(R_0(l_1, \ldots l_n)(x_0)) \sqcap x_1, \\ &\ldots, \\ &\pi_2(R_n(l_0, l_n)(x_n)) \sqcap \pi_n(R_0(l_1, \ldots l_n)(x_0)) \sqcap x_n)\end{aligned}$$

where F is defined as

$$F(x_0, x_1, \ldots, x_n) = (F_0(x1, \ldots, x_n), F_1(x_0)(x_1), \ldots, F_n(x_0)(x_n))$$

R_0 is a GRF of F_0, R_i is a GRF of $\lambda(x_0, x_i).F_i(x_0)(x_i)$ for $i \in 1 \ldots n$.[4]

Theorem 14. *A function R created using the standard reverse construction for F is an GRAST for F.*

Definition 15. *The standard reverse construction for a component abstract semantic transformer F_i is $R_i^2(l_0, l_i)(m_0, m_i) = R_i(l_0, l_i)(m_i) \sqcap (m_0, m_i)$ where R_i is a GRF for $\lambda(x_0, x_i).F_i(x_0)(x_i)$.*

Theorem 16. *A function R_i^2 creating using the standard reverse construction for a component abstract semantic transformer F_i is a GRAST for $\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$.*

[4] Note, here, R_i is a GRF for $\lambda(x_0, x_i).F_i(x_0)(x_i)$; elsewhere R_i^2 is a GRAST for $\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$.

5.4 Guided Reverse Functions for Modular Dataflow Analysis

Using the constructions developed in the previous section, it is possible to provide a general framework for dataflow analysis, which can be used to instantiate a suitably related analysis and guided reverse analysis that can be used to discover sufficient preconditions and contracts. Instantiation of the framework requires the provision an inter-component flow graph, a transfer function for node in the flow graph, and a reverse transfer function that is a GRF for each forward transfer function. We will assume that the transfer functions are associated with the flow graph to create a forward analysis and the guided reverse functions are associated with a flow graph to create a reverse analysis, but the dual is also possible.

Each node in our inter-component flow has has a unique label $\ell \in \mathbf{Lab}$ and belongs some component $i \in 1 \dots n$ or is related to some public interface. Flows internal to a component are contained in $E_{\text{local}} : \mathbf{Lab} \times \mathbf{Lab}$. Flows from interface-related nodes to nodes in some component are contained in $E_{\text{in}} : \mathbf{Lab} \times \mathbf{Lab}$. Flows from nodes in component i to some interface-related node are contained in $E_{\text{out}} : \{1, \dots, n\} \times \mathbf{Lab} \times \mathbf{Lab}$.

If the property space for the forward and reverse transfer functions ($f_\ell : M \to M$ and $r_\ell : M \to M \to M$),

then a modular analysis function $F : L_0 \times L_1 \times \dots \times L_n \to L_0 \times L_1 \times \dots \times L_n \to$ can be constructed from suitably defined F_0 and F_i. Properties in $L_0 : \mathbf{Lab} \to M$ associate interface-related nodes to properties in M and properties in $L_i : (\mathbf{Lab} \to M) \times (\mathbf{Lab} \to M)$ associate each node in component i to an entry and exit property for that component.

For the construction, we let:

$$F_0(x_{1,\circ}, x_{1,\bullet}), \dots, (x_{n,\circ}, x_{n,\bullet})) = \lambda\ell. \bigsqcup \{x_{i,\bullet}(\ell') | (i, \ell', \ell) \in E_{\text{out}}\}$$

$$\begin{aligned} &F_i(a_0, (x_{i,\circ}, x_{i,\bullet})) \\ &\quad = \Big(\lambda\ell. \bigsqcup \{x_{i,\bullet}(\ell') | (\ell', \ell) \in E_{\text{local}}\} \sqcup \bigsqcup \{a_0(\ell') | (\ell', \ell) \in E_{\text{in}}\} \sqcup \iota_\ell, \lambda\ell. f_\ell(x_{i,\circ}(\ell))\Big) \end{aligned}$$

The subscripts $\circ$ and $\bullet$ are used to designate the entry and exit properties of a given node and ι_ℓ is used to establish minimum properties at the program start or end nodes.

An R_0 which is a GRF for F_0 and an R_i which is a GRF for F_i can be constructed as follows:

$$\begin{aligned}
&R_0(l_1, \ldots, l_n)(x_0) \\
&\quad = \Big(\top, (\lambda\ell. \textstyle\bigsqcap\{x_0(\ell')|(1, \ell, \ell') \in E_{\text{out}}\}, \ldots,\ \lambda\ell. \textstyle\bigsqcap\{x_0(\ell')|(n, \ell, \ell') \in E_{\text{out}}\})\Big) \\
&R_i(l_0, (l_{i,\circ}, l_{i,\bullet}))(x_{i,\circ}, x_{i,\bullet}) \\
&\quad = \Bigg(\lambda\ell. \textstyle\bigsqcap\{x_{i,\circ}(\ell')|(\ell, \ell') \in E_{\text{in}}\}, \\
&\qquad\quad \Big(\lambda\ell. r_\ell(l_{i,\circ}(\ell))(x_{i,\bullet}(\ell)), \lambda\ell. \textstyle\bigsqcap\{x_{i,\circ}(\ell')|(\ell, \ell') \in E_{\text{local}}\}\Big)\Bigg)
\end{aligned}$$

Example 17. For programs in SMIL (Section 4, an appropriate flow graph can be define by including the each edge from a procedure call node to an export-call node (ℓ_{cn} to ℓ_{e_1}) and from a procedure exit node to an export-return nodes (ℓ_x to ℓ_{e_2}) in E_{out}, including each edges from an export-call to a procedure entry node (ℓ_{e_1} to ℓ_n) and from an export-return node to a call-site-return node (ℓ_{e_2} to $\ell_x r$) in E_{int}, and including all other edges in E_{local}.

The transfer functions and guided reverse transfer functions given in Section 4 can thus be used to instantiate a standard construction of a guided reverse dataflow analysis for SMIL programs' constant-value abstract semantics.

6 Related Work

As discussed in Section 2, there have been several prior uses of reverse abstract interpretation, especially for distributive functions (e.g., [4, 3]). In Duesterwald et al. [4], for example, reverse analysis is employed to answer queries about whether particular properties hold with out computing an entire analysis. Their approach requires that the abstract semantic transformer be distributive, so that an exact reverse of the semantic transformer can be used. For the non-distributive case, they propose using a backwards search over the flow graph to identify parts of the program's abstract domain that are relevant to solving the problem in question, and then performing a forward analysis on the program ignoring the computation of irrelevant parts of the program's property.

Besson et al. [9] use reverse abstract interpretation in the context of abstract interpretation-based proof-carrying code, where a program is annotated with additional information that is used to prove to the execution environment that the program has certain properties. The approach of [9] is to transport some witness property, which under-approximates the desired property of interest and can be shown to over-approximate the program's semantics because the program's abstract semantic transformer is reductive on the witness. Reverse analysis is used to find witnesses that are as weak (and, thus, hopefully) as small as possible. They offer a non-deterministic function for 'pruning' disjunctive clauses from witness pre-images. This is quite similar in spirit to our GRAST, and the pruning function uses the non-distributive abstract semantic transformers in order to ensure that they get a witness for their desired property that is actually an upper-approximation of the program's abstract semantics. The key difference, however, is that their pruning function is non-deterministic, where as we give requirements for defining deterministic guided reverse functions. This is expected to have a significant impact on performance.

Seo et al. [9] use a combination of (forward) abstract interpretation and Hoare logic reasoning to develop proofs in a proof-carrying code context. The abstract interpretation is used to identify loop invariants, and the Hoare logic reasoning is used to create proofs of properties of interest. Because, the loop-invariants discovered by abstract interpretation are likely to contain properties that are not relevant for showing the desired property of interest, they deploy an abstract value slicer. The abstract value slicer treat the program's abstract properties as sets of conjuncts and selects a subset of those, preserving a superset of those which might be relevant to the Hoare proof. We see this as one strategy that can be used in developing guided reverse functions, and is indeed, roughly similar to how we designed the guided-cases of our SMIL constant-value reverse transfer function for statements of the form $[x_1 = x_2 \text{ op } x_3]^\ell$ in Section 4.

Bourdoncle [10] uses a backwards abstract interpretation to find necessary preconditions for program assertions. This approach resembles ours, except that no 'guidance' is needed since necessary preconditions require an upper-approximation rather than an under-approximation of properties and upper-approximations can use the standard concretization and abstraction functions with the reverse concrete semantics. A similar approach is used in [11], which also over-approximates.

In recent work, Moy [12, 13] uses a combination of an abstract interpretation (to find loop invariants) and Hoare-style weakest-precondition calculation to find sufficient preconditions for linear-inequality assertions. Even though the abstract interpretation part of their analysis is based on a convex domain (which is non-distributive), they produces preconditions with disjunctives, which cannot be represented in a convex domain. In contrast, our guided reverse analysis uses the program's abstract semantics as guidance in choosing an appropriate pre-image, thus, inferring contracts that can be used to verify programs using convex non-distributive domains.

An approach approach similar to Moy's is taken in Chandra et al.'s Snuggle bug system [14]. This system uses backwards symbolic analysis to identify a precondition at some entry point that will ensure it reaches a goal state. It performs a path-based weakest-precondition calculation, and an ad-hoc pre-pass uses abstract interpretation to infer loop invariants. Since the symbolic analysis procedure is not guaranteed to terminate in the presence of loops and recursion, snugglebug terminates after exhausting a fixed budget. In contrast, our approach can use variants of abstract interpretations widening technique to ensure termination.

An alternative approach to contract inference, based on dynamic rather than static analysis, was pioneered by Ernst et al.'s Daikon system [15]. Daikon works based on a set of test cases that are used to compare the results of running the tool on the source code with existing data traces. It has been widely deployed in several different application contexts [16–20]. The main difference between Daikon's approach to contract inference and guided backwards analysis, is that Daikon relies on dynamic execution traces whereas guided backwards analysis is a static analysis technique. This has several implications. First of all, the approach is not sound in that it may infer spurious contracts that hold for the specific data in the execution, but not for other code paths which the program being studied might execute. Second, the contracts it infers might not be usable for performing verification using any particular abstract domain, whereas the contracts inferred by guided reverse analysis are constructed in such a way that they can be verified using the program's abstract semantics. These differences make dynamic contract inference ill-suited for contingent optimization.

The need to guide the reverse analysis ultimately stems from the abstract domain of the analysis being unable to express the weakest precondition of operations. A related problem has been addressed in the context of applying abstraction to model checking. Clark et al.'s Counter-Example Guided Abstraction Refinement (CEGAR) algorithm [21] attempts to model-check some property under an abstraction. If abstract model of the program does not satisfy the property, then the abstract counter-example is checked against the concrete system to determine whether there is a concrete counter-example or if the counter-example is spurious. When the counter-example is spurious, the algorithm repeats the model checking with a refined abstraction that excludes the spurious counter-example. Pasareanu et al.'s AGAR algorithm [22] applies this approach to find assume-guarantee rules that allow multi-component systems to be verified compositionally. This approach fundamentally differs from ours in that we work with a fixed abstract domain and find sufficient assumptions within that domain, whereas Pasareanu et al.'s work iteratively refines the domain until the weakest assumption can be represented. This refinement process can be expensive. In particular, each refinement will add an additional predicate on which the abstraction is based, and each refinement potentially requires an amount of time exponential in the number of predicates. There are several other approaches based on iterative abstraction refinement [23, 24], some of which apply it to forward and backward abstract interpretation [25–27]. All of these require an iterative abstraction refinement process. Our approach, avoids this expense, at the cost of only finding sufficient assumptions, albeit ones satisfied by the program being analyzed.

An unrelated use of the term 'guided' in the static analysis domain appears in the 'guided static analysis' of Gopan and Reps [28] Gopan and Reps. Their guided static analysis is used to reduce the loss of precision that occurs when accelerating an analysis with widening operations or through the use of a non-distributive domain. The analysis is guided to more precise solutions by analyzing the program in phases, and analyzing only a restricted version of the program in all but the last phase.

7 Discussion and Future Work

7.1 Analysis Expense

In this paper, we have assumed that a whole-program inter-modular analysis is computationally feasible. Whether this is true in practice, depends on the complexity of the abstract domain(s) used for the inter-modular analysis and the complexity of the program being analyzed. We have thought of two ways to ameliorate the analysis cost for specific applications.

First, the complexities of the abstract domains can be reduced at component boundaries. For example, it would be possible to have analysis that is context-sensitive and uses a relational numeric domain (e.g., octagons) within components, but uses only context-insensitive non-relational abstract values (e.g., intervals) across component boundaries [8, 29].

Second, since contacts can be used to verify contingent properties in changed programs, in some instances it may be possible to obtain useful contracts by examining the contingent properties of components in different program contexts. For example, it could be instructive to examine a program unit in the context of its unit test case drivers. This would allow one to derive contracts, whose violation would indicate that a program's execution is different from the program's test environment in a way related to some contingent property of interest. In the actual program context, these contracts would not automatically be verifiable using the related intra-component analysis, but static analysis and dynamic enforcement could still be applied to discover (potential) violations of the contract.

7.2 Octagon Domain

For non-distributive semantic transformers, developing an appropriate GRAST for a given forward analysis is not automatic. Although, we have illustrated how this can be accomplished for a constant-value analysis, the guided reverse functions must be designed for each new guided reverse analysis.

We have also been developing a guided reverse analysis that uses the Octagon domain [30], which relates pairs of numerical variables. There are two significant issues that make developing a guided reverse function for an octagon domain more challenging than for the constant value domain.

First, the octagon domain is of infinite height, so we have developed an acceleration that is somewhat like a guided dual of widening that can be applied to the GAST functions to ensure convergence.

Second, there are more ways a single result can be obtained, especially when considering the ways facts obtained from different sources can be combined. The octagon domain consists of sets of constraints of the form ($\pm x \pm y \leq c$), where x and y are the pair of variables related by the octagon constraint. In general, the set of octagon relationships explicitly listed in the representation of an abstract property are a subset of the relationships the property actually represents. This is because, additional relationships can be obtained by combining pairs of relations that share one variable. Therefore, octagon domain transfer functions often make use of a closure operation to discover additional octagon relations that are not explicit in an abstract property. Our reverse analysis makes use of a modified closure operation that tracks—for each octagon constraints, the pair of constraints that were combined to produce that constraint. Thus at program locations where the transfer function utilizes a closure operation, this modified closure can be applied to the guidance property to discover what constraints variables were combined to produce each constraint in the property of interest. In this way, the corresponding guided reverse function can project constraints backwards into the appropriate predecessor octagon constraints.

8 Conclusion

In a modular software system the useful properties of a component do not solely depend on the code for that component but are rather contingent upon properties of the component's environment. Therefore, if optimizations are two be performed based on these contingent properties, a mechanism is needed to determine under what conditions a contingent property will remain valid.

In this paper, we described a novel approach based on abstract interpretation that solves the problem of finding a global program property whose preservation is sufficient to ensure that the component will continue to have a certain desired behavior (property of interest) even as the rest of the program evolves. This is accomplished by performing a guided reverse analysis, which runs the analysis in reverse starting with the property of interest. Because a best reversal does not exist for non-distributive abstract semantic transformers, the guided reverse analysis must, in general, map values to an appropriate under-approximation of the pre-image of that value under the program's semantic transformer. A guidance obtained by using standard abstract interpretation techniques to the current program and is supplied to guided reverse abstract transformers during guided reverse analysis. As a result the guided reverse analysis will yield a global property that soundly approximates the current program's semantics.

We have developed a formal framework guided reverse analysis, and have defined the requirement for defining the guided reverse abstract semantic transformers and guided reverse functions needed to build and guided reverse analysis. Furthermore, we have illustrated how guided reverse functions can be constructed by defining a guided reverse analysis for the constant-value abstract semantics of programs in a simple imperative language. We have also discussed what is required to define an guided reverse analysis for abstract semantics utilizing the Octagon domain.

A Proofs of Lemmas and Theorems

Lemma 8. *Given any function R that is an GRAST for some abstract semantic transformer F over a domain L and any two properties l and m in L such that $F(l) \sqsubseteq l \sqsubseteq m$, if the sequence $((R(l))^k(m))_k$ stabilizes, the sequence will stabilize* *at a property $\bigsqcap_k((R(l))^k(m))$ of at least l, of at most m, and at which F is reductive.*[5]

Proof. Recall that the condition that R is an GRAST of F means that whenever $F(l) \sqsubseteq l \sqsubseteq m$, then the following hold:

$$R(l)\text{is monotonic} \tag{2}$$
$$F(R(l)(m)) \sqsubseteq m \tag{3}$$
$$R(l)(m) \sqsupseteq l \tag{4}$$
$$R(l)(m) \sqsubseteq m \tag{5}$$
$$\tag{6}$$

Suppose, an arbitrary complete lattice L, an arbitrary property $l : L$ and a monotonic function $F : L \to L$ and a function R that is an GRAST of F such that where $F(l) \sqsubseteq l$. To prove our lemma, we will show that:

i. For any natural number k and any property $m \sqsupseteq l$, $((R(l))^k(m)) \sqsupseteq l$
ii. For any natural number k and any property $m \sqsupseteq l$, $((R(l))^k(m)) \sqsubseteq m$
iii. For any property $m_1 \sqsupseteq l$, any property $m_2 = \bigsqcap_k((R(l))^k(m_1))$ at which $((R(m_1))^k)_k$ stabilzes is such that:
   a. $F(m_2) \sqsubseteq m_2$
   b. $l \sqsubseteq m_2 \sqsubseteq m_1$

The proof of part (i), $\forall k, m : m \sqsupseteq l \Rightarrow ((R(l))^k(m)) \sqsupseteq l$, proceeds by induction on k. For the base case, observe that $((R(l))^0(m) = m$. The inductive case can be shown using (4) with $(R(l))^{i-1}(m)$ as m.

The proof of part (ii), $\forall k, m : m \sqsupseteq l \Rightarrow ((R(l))^k(m)) \sqsubseteq m$, proceeds by induction on k. The base case follows directly from (5). The inductive case follows from part i with $k-1$ as k and (5) with $(R(l))^{k-1}(m)$ as m.

For the final part of the proof, first, suppose an arbitrary $m_1 \sqsupseteq l$, and let m_2 be $m_2 = \bigsqcap_k((R(l))^k(m_1))$. Furthermore, if the sequence $((R(m_1))^k)_k$ stabilizes, there exists a k, such that $R(l)((R(l))^k(m)) = (R(l))^k(m_1) = m_2$., and so $R(l)(m_2) = m_2$. Since m_2 has the form $(R(l))^k(m_1)$, we know $l \sqsubseteq m_2 \sqsubseteq m_1$ from parts i and ii. Consequently, from (3), we have that $F(R(l)(m_2)) \sqsubseteq m_2$, and thus $F(m_2) \sqsubseteq m_2$. That is F is reductive at m_2.

Theorem 9. *If R_i^2 is a GRAST for $\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$ where F_i is the abstract semantic function of some component and $(l_i) \sqsupseteq F_i(l_0)(l_i)$, then the assumption a found by computing:*[6]

$$a = \pi_1\left(\bigsqcap_k (R_i^2(l_0, l_i))^k(\top, m_i)\right)$$

[5] We use the notiation $(t(k))_k$ to denote the infinite sequence $t(0), t(1), \ldots$ where $t(k)$ stands for an arbitrary term with the free variable of k.

[6] We use the notation $\pi_n(e)$, where e is an expression evaluating to a tuple value, to denote the nth element (starting at 1) of that tuple.

for some property of interest $m_i \sqsupseteq l_i$ is sufficiently strong to guarantee that, for any global property $x_0 \sqsubseteq a$:

$$m_i \sqsupseteq \mathrm{lfp}(F_i(x_0))$$

whenever the sequence $((R_i^2(l_0, l_i))^k(\top, m_i))_k$ converges.

Proof. By Lemma 8:

$$(\top, m_i) \sqsupseteq (a, g_i) \sqsupseteq (\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i)))(a, g_i) = (a, F_i(a)(g_i))$$

and so,

$$m_i \sqsupseteq g_i \sqsupseteq F_i(a)(g_i)$$

Furthermore, for all $x_0 \sqsubseteq a$:

$$m_i \sqsupseteq g_i \sqsupseteq F_i(x_0)(g_i)$$

since F_i is a monotonic.

Furthermore, since $F_i(x_0)$ is a monotonic:

$$\mathrm{lfp}(F_i(x_0)) \sqsubseteq g_i \sqsubseteq m_i$$

Theorem 12. *If $(a_{1,0}, g_{1,1}) \dots (a_{n,0}, g_{n,n})$ is a set of consistent local contracts for a property of interest $(m_0, m_1, \dots, m_n)$ of a program described by a modular abstract semantic transformer F, then:*

1. *the local guarantee for each component i holds if the corresponding local assumption hold:*

$$g_{i,i} \sqsupseteq \mathrm{lfp}(F_i(a_{i,0}))$$

2. *the guarantees are sufficient to establish each of the assumptions:*

$$a_{i,0} \sqsupseteq F_0(g_{1,1}, \dots g_{n,n}) \text{ for } i \in 1 \dots n$$

3. *the property of interest holds if all of the local assumptions hold (and hence all of the guarantees hold)*

$$\begin{aligned} &m_0 \sqsupseteq F_0(g_{1,1}, \dots g_{n,n}) \\ &m_i \sqsupseteq g_{i,i} \end{aligned}$$

Proof. Since $(a_{1,0}, g_{1,1}) \dots (a_{n,0}, g_{n,n})$ is *a set of consistent local contracts,* they were derived by calculating:

$$(a_{i,0}, g_{i,i}) = \bigsqcap_k ((R_i^2(l_0, l_i))^k(m_0, g_{0,i}))$$

using $g_{0,i}$ taken from *globally sufficient assumptions and guarantees* calculated with:

$$(a_{0,0}, g_{0,1}, \dots, g_{0,n}) = \bigsqcap_k R(l_0, l_1, \dots, l_n)^k(m_0, m_1, \dots, m_n)$$

We first show that, for arbitrary $i \in 1 \ldots n$:

$$(a_{0,0}, g_{0,i}) \sqsubseteq (a_{i,0}, g_{i,i}) \sqsubseteq (\top, g_{0,i})$$

Because $(a_{i,0}, g_{i,i})$ was computed using an GRAST with $(\top, g_{0,i})$ as the desired property property of interest and (l_0, l_i) as its guidance, we know from Lemma 8 that $(l_0, l_i) \sqsubseteq (a_{i,0}, g_{i,i}) \sqsubseteq (\top, g_{0,i})$. It follow that $(a_{0,0}, g_{0,i})$ is a member of the sublattice $\{(x_0, x_i) \sqsubseteq L_0 \times L_i \mid (l_0, l_i) \sqsubseteq (\top, g_{0,i})\}$. Since $Ł_0 \times Ł_i$ is a complete lattice, this sublattice is also a complete lattice. Because of how R is defined in terms of R_i^2, and because $(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$ is a fixed point of R, it follows that $(a_{0,0}, g_{0,i})$ is a fixed point of $R_i^2(l_0, l_i)$ for $i \in 1 \ldots n$. Consequently, by the dual of Kleene's fixed point theorem and Lemma 8, the limit of the sequence $((R_i^2(l_0, l_i))^k(\top, g_{0,i}))_k$ must be the greatest fixed point in the sublattice. Since $(a_{0,0}, g_{0,i})$ is also a fixed point, this limit $(a_{i,0}, g_{i,i}) = \bigsqcap_k((R_i^2(l_0, l_i))^k(m_0, g_{0,i}))$ must be at least $(a_{0,0}, g_{0,i})$. Additionally, because $(a_{0,0}, g_{0,i}) \sqsubseteq (a_{i,0}, g_{i,i}) \sqsubseteq (\top, g_{0,i})$, if follows that $a_{0,0} \sqsubseteq a_{i,0}$ and $g_{0,i} = g_{i,i}$ for $i \in 1 \ldots n$.

To show that $g_{i,i} \sqsupseteq \text{lfp}(F_i(a_{i,0}))$, recall that by Lemma 8, $(a_{i,0}, g_{i,i}) \sqsupseteq (\lambda(x_0, x_i).(x_0, F_i(x_0, x_i)))(a_{i,0}, g_{i,i}) = (a_{i,0}, F(a_{i,0})(g_{i,i}))$ and so $g_{i,i} \sqsupseteq F(a_{i,0})(g_{i,i})$ and by Tarski's Fixed Point Theorem:

$$g_{i,i} \sqsupseteq \text{lfp}(F_i(a_{i,0}))$$

We now show $a_{i,0} \sqsupseteq F_0(g_{0,1}, \ldots g_{0,n})$ and, consequently, since $g_{0,i} = g_{i,i}$, that $a_{i,0} \sqsupseteq F_0(g_{1,1}, \ldots g_{n,n})$. Because of Lemma 8 and that $(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$ are globally sufficient assumptions and guarantees, we know that:

$$(a_{0,0}, g_{0,1}, \ldots, g_{0,n}) \sqsupseteq F(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$$

By the construction of F from F_0 and F_i and because $a_{i,0} \sqsupseteq a_{0,0}$, it follows that:

$$a_{i,0} \sqsupseteq a_{0,0} \sqsupseteq F_0(g_{0,1}, \ldots, g_{0,n}) = F_0(g_{1,1}, \ldots g_{n,n})$$

for $i \in 1 \ldots n$.

We now show that $m_0 \sqsupseteq F_0(g_{0,1}, \ldots g_{0,n})$. and, since $g_{0,i} = g_{i,i}$, that $m_0 \sqsupseteq F_0(g_{1,1}, \ldots g_{n,n})$. Because of Lemma 8 and that $(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$ are globally sufficient assumptions and guarantees, we know that:

$$(a_{0,0}, g_{0,1}, \ldots, g_{0,n}) \sqsupseteq F(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$$

and by the construction of F from F_0 and F_i, we know that:

$$a_{0,0} \sqsupseteq F_0(g_{0,1}, \ldots, g_{0,n})$$

That $m_0 \sqsupseteq a_{0,0}$ follows from the construction of $(a_{0,0}, g_{0,1}, \ldots, g_{0,n})$ as a lower bound of the descending chain starting at $(m_0, m_1, \ldots, m_n)$. And so,

$$m_0 \sqsupseteq a_{0,0} \sqsupseteq F_0(g_{0,1}, \ldots, g_{0,n})$$

That $m_i \sqsupseteq g_{i,i}$, for arbitrary $i \in 1 \ldots n$, follows from the equality $g_{i,i} = g_{0,i}$, Lemma 8 and the construction of $g_{0,i}$ as a component of globally sufficient assumption and guarantees.

Theorem 14 *A function R created using the standard reverse construction for F is an GRAST for F.*

Proof. Suppose a modular abstract function $F : L \to L$, an arbitrary guidance property $(l_0, l_1, \ldots, l_n) : L$ and desired property of interest $(m_0, m_1, \ldots, m_n) : L$, such that $F(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n) \sqsubseteq (l_0, l_1, \ldots, l_n) \sqsubseteq (m_0, m_1, \ldots, m_n)$, and a function R created using the standard reverse construction for F.

That R is an GRAST for F can be proven by showing:

i. R is monotonic
ii. $F(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n)) \sqsubseteq (m_0, m_1, \ldots, m_n)$,
iii. $R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n) \sqsupseteq (l_0, l_1, \ldots, l_n)$
iv. $R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n) \sqsubseteq (m_0, m_1, \ldots, m_n)$

That R is monotonic (part i) follows from the definition of R and the monotonicity of R_0 and each R_i.

That $F(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n)) \sqsubseteq (m_0, m_1, \ldots, m_n)$ (part ii) can be established based by showing:

a. $\pi_i(F(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n))) \sqsubseteq m_0$ and
b. $\pi_{i+1}(F(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n))) \sqsubseteq m_i$ for each $i \in 1...n$.

To show ii(a), we compute

$$\begin{aligned}
&\pi_1(F(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n))) \\
&\quad = \pi_1 \left(F \left(\pi_1 \left(\prod_{i=1}^{n} (R_i(l_0, l_i)(m_i)) \right) \sqcap m_0 \right), \right. \\
&\qquad\quad \pi_2(R_1(l_0, l_1)(m_1)) \sqcap \pi_1(R_0(l_1, \ldots, l_n)(m_0)) \sqcap m_1, \\
&\qquad\quad \ldots, \\
&\qquad\quad \left. \pi_2(R_n(l_0, l_1)(m_n)) \sqcap \pi_n(R_0(l_1, \ldots, l_n)(m_0)) \sqcap m_n \right) \\
&\quad = F_0(\pi_2(R_1(l_0, l_1)(m_1)) \sqcap \pi_1(R_0(l_1, \ldots, l_n)(m_0)) \sqcap m_1, \\
&\qquad\quad \ldots, \\
&\qquad\quad \pi_2(R_n(l_0, l_1)(m_n)) \sqcap \pi_n(R_0(l_1, \ldots, l_n)(m_0)) \sqcap m_n) \qquad \sqsubseteq F_0(R_0((l_0, l_1, \ldots, l_n)(m_0))) \\
&\quad \sqsubseteq m_0
\end{aligned}$$

using that R_0 is GRF of F_0.

To show part ii(b), we compute for an arbitrary $i \in 1 \ldots n$:

$$\begin{aligned}
&\pi_{i+1}(F(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n))) \\
&\quad = F_i(\pi_1(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n)), \pi_2(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n))) \\
&\quad = F_i \left(\prod_{i=1}^{n} (\pi_1(R_i(l_0, l_i)(m_i))), \pi_2(R_i(l_0, l_i)(m_i)) \sqcap \pi_i(R_0(l_1, \ldots, l_n)(m_0)) \sqcap m_i \right) \\
&\quad \sqsubseteq F_i(\pi_1(R_i(l_0, l_i)(m_i)), \pi_2(R_i(l_0, l_i)(m_i))) \\
&\quad = F_i(R_i(l_0, l_i)(m_i)) \\
&\quad \sqsubseteq m_i
\end{aligned}$$

using R_i is GRF for $\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$.

That $R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n) \sqsupseteq (l_0, l_1, \ldots, l_n)$ (part iii) can be established by showing

a. $\pi_1(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n)) \sqsupseteq l_0$ and
b. $\pi_{i+1}(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n)) \sqsupseteq l_i$

To show iii(a), we compute

$$\begin{aligned}
\pi_1(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n)) &= \left(\prod_{i=1}^{n}(\pi_1(R_i(l_0, l_i)(m_i)))\right) \sqcap m_0 \\
&\sqsupseteq \left(\prod_{i=1}^{n} \pi_1(l_0, l_n)\right) \sqcap m_0 \\
&= \pi_1(l_0, l_n) \sqcap m_0 \\
&= l_0 \sqcap m_0 \\
&= l_0
\end{aligned}$$

using that $R_i(l_0, l_i)(m_i) \sqsupseteq (l_0, l_i)$ because R_i is a GRF for $\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i))$ and (l_0, l_i) is a valid guidance for m_i.

To show iii(b), we compute

$$\begin{aligned}
\pi_{i+1}(R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n)) &= \pi_2(R_i(l_0, l_i)(m_i)) \sqcap \pi_i(R_0(l_1, \ldots, l_n)(m_0)) \sqcap m_i \\
&\sqsupseteq pi_2(l_0, l_i) \sqcap \pi_i(l_1 \ldots l_n) \sqcap m_i \\
&\sqsupseteq l_i \sqcap l_i \sqcap m_i \\
&= l_i
\end{aligned}$$

using both that $R_i(l_0, l_i)(m_i) \sqsupseteq (l_0, l_i)$ because R_i is a GRF and (l_0, l_i) is a valid guidance property for m_i with respect to $\lambda(x_0, x_i).F_i(x_0)(x_i)$, and also that $R_0(l_1, \ldots, l_n)(m_0) \sqsupseteq (l_1, \ldots, l_n)$ because R_0 is a GRF for F_0, $(l_1, \ldots, l_n)$ is a valid guidance with respect to F_0.

That $R(l_0, l_1, \ldots, l_n)(m_0, m_1, \ldots, m_n) \sqsubseteq (m_0, m_1, \ldots, m_n)$ (part iv) follows directly from the definition of R.

Theorem 16. *A function R_i^2 creating using the standard backwards construction for a component abstract semantic transformer F_i is a GRAST for $\lambda(m_0, m_i).(m_0, F_i(m_0)(m_i))$.*

Proof. To prove that R_i^2 is GRAST for $\lambda(m_0, m_i).(m_0, F_i(m_0)(m_i))$, we need to show:

i. $R_i^2(l_0, l_i)$ is monotonic
ii. $(\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i)))(R_i^2(l_0, l_i)(m_0, m_i)) \sqsubseteq (m_0, m_i)$,
iii. $R_i^2(l_0, l_i)(m_0, m_i) \sqsupseteq (l_0, l_i)$, and
iv. $R_i^2(l_0, l_i)(m_0, m_i) \sqsubseteq (m_0, m_i)$

with an upper bound being guaranteed by the hypothesis.

Part (i) follows from the definition of R_i^2 and the monotonicity of R_i because R_i is a GRF for $\lambda(x_0, x_i).F(x_0)(x_i)$.

Part (ii) follows from the definition of R_i^2 and from the proposition that R_i is GRF for $\lambda(x_0, x_i).F_i(x_0)(x_i)$ (which means that $(m_0, F_i(R_i(l_0, l_i)(m_i)) \sqsubseteq (m_0, m_i))$ by computing

$$(\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i)))(R_i^2(l_0, l_i)(m_0, m_i)) = (\lambda(x_0, x_i).(x_0, F_i(x_0)(x_i)))(R_i(l_0, l_i)(m_i) \sqcap (m_0, m_i)) \quad = (\pi_1($$

Part (iii) follows from the hypothesis that $(l_0, l_i) \sqsubseteq (m_0, m_i)$ and the definition of R_i^2 by computing

$$\begin{aligned} R_i^2(l_0, l_i)(m_0, m_i) &= R_i(l_0, l_i)(m_i) \sqcap (m_0, m_i) \\ &\sqsupseteq (l_0, l_i). \end{aligned}$$

Part (iv) follows from the definition of R_i^2 by computing

$$\begin{aligned} R_i^2(l_0, l_i)(m_0, m_i) &= R_i(l_0, l_i)(m_i) \sqcap (m_0, m_i) \\ &\sqsubseteq (m_0, m_i) \end{aligned}$$

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (Jan 1977)
2. Dijkstra, E.W.: Guarded commands, non-determinancy and a calculus for the derivation of programs. In Bauer, F.L., Samelson, K., eds.: Language Hierarchies and Interfaces. Volume 46 of Lecture Notes in Computer Science., Springer (1975) 111–124
3. Hughes, J., Launchbury, J.: Reversing abstract interpretations. In: Symposium proceedings on 4th European symposium on programming, London, UK, Springer-Verlag (1992) 269–286
4. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. ACM Trans. Program. Lang. Syst. **19** (November 1997) 992–1030
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '79, New York, NY, USA, ACM (1979) 269–282
6. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
7. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science **277**(1-2) (2002) 47 – 103
8. Cousot, P., Cousot, R.: Modular static program analysis. In Horspool, R.N., ed.: CC. Volume 2304 of Lecture Notes in Computer Science., Springer (2002) 159–178
9. Besson, F., Jensen, T., Turpin, T.: Small witnesses for abstract interpretation-based proofs. In De Nicola, R., ed.: Programming Languages and Systems. Volume 4421 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 268–283 10.1007/978-3-540-71316-6_19.

10. Bourdoncle, F.: Assertion-based debugging of imperative programs by abstract interpretation. In Sommerville, I., Paul, M., eds.: ESEC. Volume 717 of Lecture Notes in Computer Science., Springer (1993) 501–516
11. Cousot, P., Cousot, R., Logozzo, F.: Precondition inference from intermittent assertions and application to contracts on collections. In Jhala, R., Schmidt, D., eds.: Verification, Model Checking, and Abstract Interpretation. Volume 6538 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2011) 150–168 10.1007/978-3-642-18275-4_12.
12. Moy, Y.: Sufficient preconditions for modular assertion checking. In Logozzo, F., Peled, D., Zuck, L., eds.: Verification, Model Checking, and Abstract Interpretation. Volume 4905 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 188–202
13. Moy, Y., Marché, C.: Modular inference of subprogram contracts for safety checking. Journal of Symbolic Computation **45**(11) (2010) 1184 – 1211 Special Issue on Invariant Generation and Advanced Techniques for Reasoning about Loops.
14. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In Hind, M., Diwan, A., eds.: PLDI, ACM (2009) 363–374
15. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3) (2007) 35–45
16. McCamant, S., Ernst, M.: Formalizing lightweight verification of software component composition. SAVCBS 2004 Specification and Verification of Component-Based Systems (2004) 47
17. McCamant, S., Ernst, M.D.: Early identification of incompatibilities in multi-component upgrades. In Odersky, M., ed.: ECOOP. Volume 3086 of Lecture Notes in Computer Science., Springer (2004) 440–464
18. Perkins, J.H., Ernst, M.D.: Efficient incremental algorithms for dynamic detection of likely invariants. In Taylor, R.N., Dwyer, M.B., eds.: SIGSOFT FSE, ACM (2004) 23–32
19. Nimmer, J.W., Ernst, M.D.: Invariant inference for static checking. In: SIGSOFT FSE. (2002) 11–20
20. Nimmer, J.W., Ernst, M.D.: Static verification of dynamically detected program invariants: Integrating daikon and esc/java. Electr. Notes Theor. Comput. Sci. **55**(2) (2001)
21. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In Emerson, E.A., Sistla, A.P., eds.: CAV. Volume 1855 of Lecture Notes in Computer Science., Springer (2000) 154–169
22. Bobaru, M.G., Pasareanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 135–148
23. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. ESEC/FSE-13, New York, NY, USA, ACM (2005) 31–40
24. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design **32**(3) (2008) 175–205
25. Cousot, P., Ganty, P., Raskin, J.F.: Fixpoint-guided abstraction refinements. In Nielson, H.R., Filé, G., eds.: SAS. Volume 4634 of Lecture Notes in Computer Science., Springer (2007) 333–348

26. Ranzato, F., Rossi-Doria, O., Tapparo, F.: A forward-backward abstraction refinement algorithm. In Logozzo, F., Peled, D., Zuck, L.D., eds.: VMCAI. Volume 4905 of Lecture Notes in Computer Science., Springer (2008) 248–262
27. Massé, D.: Combining forward and backward analyses of temporal properties. In Danvy, O., Filinski, A., eds.: PADO. Volume 2053 of Lecture Notes in Computer Science., Springer (2001) 103–116
28. Gopan, D., Reps, T.W.: Guided static analysis. In Nielson, H.R., Filé, G., eds.: SAS. Volume 4634 of Lecture Notes in Computer Science., Springer (2007) 349–365
29. Sands, D., ed.: Typestate Checking of Machine Code. In Sands, D., ed.: ESOP. Volume 2028 of Lecture Notes in Computer Science., Springer (2001)
30. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1) (2006) 31–100