Semantic Foundations for the Static Analysis of Program Revisions

DAKOTA BRYAN, University of Colorado Boulder, USA

1 Introduction

During the software development process, programmers create pull requests (PRs) to modify the behavior of a system. These PRs are subsequently reviewed to determine if the changes introduced to the code produce the intended consequences. Often during these reviews, informal notions of the semantics of the change are used. For example, "in the old version the value of `x` could be `10`, but the new version does not allow that." Or "the only states that we're removed are one's where the value of `x` is `17`". Or even, "in the states when the values of `x` is the same in both the new and old version, the values of `y` are different". In this work, we present a concrete semantics that can describe the semantic effect of program revision, and an abstraction interpretation over it which enables reasoning akin the examples above.

There are many program analyses that enable reasoning about two programs; however, we claim that they fall short by not capturing the mental model developers hold of diffs and not bringing that model to the forefront of the analysis. One existing technique, "Abstract Semantic Differencing for Numerical Programs" [9] is designed specifically for program differences, and has a concept of "added", "removed", and "related" states. However, we claim their version of determining if a variable value is removed does not capture all ways that a programmer may consider a value to be removed. It is fundamentally based on a state-to-state comparison (akin to relational Hoare logic) which does not allow for the set of "added" states to have a different cardinality as the set of "removed" states.

We center our analysis around a characterization of "old" and "new" behaviors as added, removed, or related. In order to enable the possibility for the added and removed set of behaviors to have a different cardinality, we must form this partition via a state-to-set of state comparison. The checks needed to form this partition of possible states is not always obvious enough to encode with RHL. Existing analyses which can form a similar partition do not provide a natural way to write assertions over them, or shape the analysis around them.

1.1 Contributions

Our main contributions can be summarized as follows:

- We first give a concrete semantics that describes the semantic effect of program revision.
- Then, we define an abstract semantics over this, which allows for proofs about program revisions.

2 Overview

In this section, we work through an example program revision and prove differential properties about the revision. We see how our natural intuitions about program differences lead to the formalisms and generalizations given in section 3. Particularly, we see that a natural way to describe program differences leads to the concrete semantics of our "diff machine" given in section 3.2. Furthermore, we see that this "diff machine" must be parametrized by some notion of state equivalence, which generalizes to the *state correspondence* given in section 3.1. Finally, in an attempt

Author's Contact Information: Dakota Bryan, University of Colorado Boulder, Boulder, Colorado, USA, dakota.bryan@colorado.edu.

Listing 1. (Sign Program)

```
x := nonDet()
if (x > 0)
    sgn := 1
else
    sgn := -1
```

Listing 2. (Patched Sign Program)

```
x := nonDet()
if (x > 0)
    sgn := 1
else
    sgn := -1
if (x = 0)
    sgn := 0
```

Fig. 1. An example revision to a program that computes the sign of a number

to prove safety properties of the reachable "diff machine" states, we notice several intricacies which are crucial to the design of our parameterized abstract interpreter (section 3.5).

We will consider imperative programs written in the IMP language with non-deterministic assignment (the grammar can be found in appendix A).

Consider the program and corresponding revision given in figure 1. We will let program variables subscripted with $-$ denote variables in the "old" program and $+$ subscripts for "new" program variables. For example, x_- refers to x in program 1. We will also let Σ_- denote the reachable states of program 1 and Σ_+ denote the set of reachable states for program 2. We see that the unrevised program (1) computes the sign of x_- with zero considered negative. The revised program (2) adds the functionality that the sign of zero is zero.

2.1 Describing the Semantic Effect of Program Revision

Before proving properties about program revisions, we need a way to represent revisions; then, we can prove properties over that representation. Therefore, our first task is to describe the semantic effect of a revision (this will generalize to our concrete semantics). As we saw in the section 1, software developers can view a program revision in terms of the behavior which it removed, added, and kept the same. For IMP programs, one measure of behavior is the set of reachable states. Thus, a first attempt of describing the semantic effect of a revision is characterizing reachable states of the two programs as removed, related, or added. To form the removed set, **rm**, we could check if some reachable state of the old program is not reachable in the new program, and if so, that state would be removed. This set will be considered the behavior that the revision removed.

Now, considering our example and this approach, we would have $\sigma_- := \mathsf{x} \mapsto 0, \mathsf{sgn} \mapsto -1$ as a removed state. However, one may view the effect of the revision as adding the functionality where the sign of a number can now be zero. With this view, the revision removes no behavior. In general, the set of removed states should need not have the same cardinality as the set of added states. For example, a revision that removes bugs but adds no functionality is naturally modeled with **add** (the set of "added" behavior) empty and **rm** filled with the buggy states.

In order to address this, we can consider a σ_- to be removed only if its value of sgn is not a possible value of sgn in Σ_+ ($\mathbf{rm} := \{\sigma_- \in \Sigma_- \mid \forall \sigma_+ \in \Sigma_+.\ \sigma_-\langle \mathsf{sgn}\rangle \neq \sigma_+\langle \mathsf{sgn}\rangle\}$). Thus, whereas before we built $\mathbf{rm}$ by checking $\sigma_- \not\equiv \sigma_+$ for every $\sigma_+ \in \Sigma_+$, we now check some projection of the two states. So, with this updated definition, we have $\mathbf{rm} := \{\}$. We can compute $\mathbf{add}$ the same way to achieve $\mathbf{add} := \{\mathsf{x} \mapsto 0, \mathsf{sgn} \mapsto 0\}$.

Now, what about behavior that the revision kept the same? We first notice that if there is some $\sigma_- \notin \mathbf{rm}$, then there must be some $\sigma_+ \in \Sigma_+$ where $\sigma_-\langle \mathsf{sgn}\rangle = \sigma_+\langle \mathsf{sgn}\rangle$. This implies that each reachable old state is either removed or *related* to some reachable state for the new program, when the relation used to compare them is the state projection check. This "projection check" will henceforth be referred to as the *state correspondence*. Now, for checking properties over the semantic effect of this revision, it is natural to check properties about the behavior that is not removed and not added. So, as a first guess of defining the "related", say $\mathbf{rel}$, part of our "diff machine" (*i.e.* semantic description of the effect of the revision, *i.e.* concrete semantics), we let $\mathbf{rel} := (\Sigma_- \setminus \mathbf{rm}) \times (\Sigma_+ \setminus \mathbf{add})$.

This construction certainly captures the behaviors that were not removed or added, but we would have the following state pair in $\mathbf{rm}$: $(\mathsf{x}_- \mapsto 5, \mathsf{sgn}_- \mapsto 1), (\mathsf{x}_+ \mapsto -5, \mathsf{sgn}_+ \mapsto -1)$. This state pair isn't exactly *related*. Instead, we want every $(\sigma_-, \sigma_+) \in \mathbf{rel}$ to be related, or satisfy the state correspondence: $\mathsf{sgn}_- = \mathsf{sgn}_+$. This is now a little more natural; we could prove that in all related states, if $\mathsf{x}_- > 0$, then $\mathsf{x}_+ > 0$. This alternate formulation of $\mathbf{rel}$ is the same as $(\Sigma_- \setminus \mathbf{rm}) \times (\Sigma_+ \setminus \mathbf{add})$ with the state correspondence added as an implication antecedent to every assertion over it. This new formulation still characterizes the entire reachable state space such that $\forall \sigma_- \in \Sigma_-.\ \sigma_- \in \mathbf{rm} \vee \exists \sigma_+.(\sigma_-, \sigma_+) \in \mathbf{rel}$. We "leave nothing out".

2.2 Reasoning About Program Revisions

Equipped with the description of the semantic effect of a program revision $(\mathbf{rm}, \mathbf{rel}, \mathbf{add})$, we turn our attention to proving safety properties over this description. We will continue with the program revision shown in figure 1, along with the particular $\mathbf{rm}, \mathbf{rel}, \mathbf{add}$ for that program revision explained in section 2.1.

Consider $\varphi_{\mathbf{rm}}$ as a safety property over $\mathbf{rm}$; so, $\varphi_{\mathbf{rm}}$ relies on old program variables ($\varphi_{\mathbf{rm}}(\sigma_-)$). And $\varphi_{\mathbf{rel}}$ as a safety property over $\mathbf{rel}$, which is a relation property over two states ($\varphi_{\mathbf{rel}}(\sigma_-, \sigma_+)$). Now, we have $(\varphi_{\mathbf{rm}}, \varphi_{\mathbf{rel}}, \varphi_{\mathbf{add}})$ as a safety property over $\mathbf{rm}, \mathbf{rel}, \mathbf{add}$. We notice that $\mathbf{rm}, \mathbf{rel}, \mathbf{add}$ is a certain filtered partition of $\Sigma_+ \times \Sigma_+$; so, we will attempt to use relational Hoare logic to prove $(\varphi_{\mathbf{rm}}, \varphi_{\mathbf{rel}}, \varphi_{\mathbf{add}})$.

Recall the semantics of relational Hoare logic, shown below.

$$\models \{\varphi\}\ s_- \pm s_+\ \{\varphi'\} \text{ if and only if}$$
$$[(\sigma_-, \sigma_+) \models \varphi \text{ and } \sigma_- \vdash s_- \Downarrow \sigma_-' \text{ and } \sigma_+ \vdash s_+ \Downarrow \sigma_+] \text{ implies } (\sigma_-', \sigma_+') \models \varphi'$$

Let $\varphi_{\mathbf{add}} := \mathsf{x}_+ = 0$, $\varphi_{\mathbf{rel}} := \mathsf{x}_- > 0 \implies \mathsf{x}_+ > 0$, $s_- :=$ program 1, and $s_+ :=$ program 2. Let's first attempt to prove $\varphi_{\mathbf{rel}}$; recall, $(\sigma_-, \sigma_+) \in \mathbf{rel}$ if and only if $\sigma_-\langle \mathsf{sgn}\rangle = \sigma_+\langle \mathsf{sgn}\rangle$. So, we can build the post condition of our relational Hoare triple by first filtering state pairs which satisfy our state correspondence, then checking if those satisfy $\varphi_{\mathbf{rel}}$. $\varphi' := \mathsf{sgn}_- = \mathsf{sgn}_+ \implies \varphi_{\mathbf{rel}}$. Great, $\{\top\}\ s_- \pm s_-\ \{\varphi'\}$ is derivable so we can prove our assertion over $\mathbf{rel}$, indicating that safety properties about behaviors which a revision preserves can be proven using relational Hoare logic.

Now, let's attempt to prove $\varphi_{\mathbf{add}}$ over $\mathbf{add}$. Recall, $\sigma_+ \in \mathbf{add}$ if and only if, for all $\sigma_- \in \Sigma_-$, $\sigma_+\langle \mathsf{sgn}\rangle \neq \sigma_-\langle \mathsf{sgn}\rangle$. How can we encode that the state correspondence does not hold for any σ_- to use as our filtering antecedent? Post conditions in relational Hoare logic are interpreted over all reachable state pairs. So if we wrote $\mathsf{sgn} \neq \mathsf{sgn}$ in the post condition, it would be satisfied by $(\sigma_- = \mathsf{x}_- \mapsto 5, \text{-}sgn \mapsto 1), (\sigma_+ = \mathsf{x}_+ \mapsto -5, \mathsf{sgn}+ \mapsto -1)$. This is not what we want. In fact, there is no way to encode our property into relational Hoare logic without oracle power to determine the set of sgn values in the old program. This is because our property is *hyper*, for one new state, we need to compare it against the *set* of reachable old states. Relational Hoare logic (or Hoare logic) assertions must be over *states*, not *sets of states*.

The fact that related properties can be proven in relational Hoare logic but not added or removed properties is an indication as to why much attention has been placed on proving *equivalence* of program revisions, but not *difference*. Furthermore, to ensure that $|\mathbf{rm}| \neq |\mathbf{add}|$ it is insufficient to compare on the state level. If the two states were found to be not related, then the old one would go in **rm** and the new one in **add**, guaranteeing the sets to be equal cardinality.

Now, let's take a slightly different approach. We can use Hoare logic to find a formula that represents all possible sign values in the old program. Consider the derivable Hoare logic triple, $\{\top\}\ s_-\ \{\varphi_-\}$ where $\varphi_- = (\mathsf{x}_- > 0 \implies \mathsf{sgn}_- = 1) \land (\mathsf{x}_- \leq 0 \implies \mathsf{sgn}_- = -1)$. φ_- contains all the information we need for the possible values of sgn_-; so, we can use this in combination with a formula representing reachable states of the new program (say φ_+) to prove $\varphi_{\mathbf{add}}$. We just need some formula that checks if $\mathsf{sgn}_- \neq \mathsf{sgn}_+$ holds for all $\sigma_- \in \Sigma_-$, given φ_- that represents Σ_-. We can universally quantify over variables modified by program 1 to achieve $\forall \mathsf{x}_-, \mathsf{sgn}_-.\ \varphi_-$ which eliminates φ_-'s dependence of the variables in program 1. This has the effect of transforming φ_- into a *global* property (from a state property) that must hold for every possible assignment of those variables; which is exactly what we desire.

Now, we can add φ_- as the antecedent to $\mathsf{sgn}_- \neq \mathsf{sgn}_+$ in a formula that universally quantifies over x_- and sgn_- to achieve a formula which represents all old states such that $\mathsf{sgn}_- \neq \mathsf{sgn}_+$ holds for possible values of sgn in the old program, shown below.

$$\forall \mathsf{x}_-, \mathsf{sgn}_-.\ (\varphi_- \implies \mathsf{sgn}_- \neq \mathsf{sgn}_+)$$

Just as the quantification removes φ_-'s dependence on σ_-, it also the dependence on σ_- for $\mathsf{sgn}_- \neq \mathsf{sgn}_+$. Thus, the entire formula is interpreted over σ_+, as desired. This formula implies $\varphi_r := 1 \neq \mathsf{sgn}_+ \land -1 \neq \mathsf{sgn}_+$, which describes all old states that satisfy our removed state condition.

Now, given the derivable Hoare triple $\{\top\}\ s_+\ \{\varphi_+\}$ where $\varphi_+ = (\mathsf{x}_+ > 0 \implies \mathsf{sgn}_+ = 1) \land (\mathsf{x}_+ = 0 \implies \mathsf{sgn}_+ = 0) \land (\mathsf{x}_+ < 0 \implies \mathsf{sgn}_+ = -1)$, we can create $\varphi_+ \land \varphi_r$. This has the effect of filtering states described by φ_+ (reachable old program states) to only those that that satisfy φ_r (states that satisfy our removed state condition). And notice $\varphi_+ \land \varphi_r \implies \mathsf{x}_+ = 0$, which was our desired safety assertion over **add**. Perfect, we have proven our **add** property; however, we notice the following issue. Since Hoare logic is an over approximating program logic, our old program triple, $\{\top\}\ s_-\ \{\varphi_-\}$ where $\varphi_- = (\mathsf{x}_- > 0 \implies \mathsf{sgn}_- = 1) \land (\mathsf{x}_- \leq 0 \implies \mathsf{sgn}_- = -1)$, implies the triple $\{\top\}\ s_-\ \{\varphi_- \lor \mathsf{sgn}_- = 0\}$. Then, $\forall \mathsf{x}_-, \mathsf{sgn}_-.\ ((\varphi_- \lor \mathsf{sgn}_- = 0) \implies \mathsf{sgn}_- \neq \mathsf{sgn}_+)$ implies $\varphi_r := 1 \neq \mathsf{sgn}_+ \land -1 \neq \mathsf{sgn}_+ \land 0 \neq \mathsf{sgn}_-$. Now, we add the φ_+ conjunct ($\varphi_+ \land \varphi_r$), but this implies $\bot$. $\bot$ is certainly not a sound over approximation of **add**.

Let's go back to our definition of added states, but consider approximations, $\widehat{\mathbf{add}} := \{\sigma_+ \in \hat{\Sigma}_+ \mid \forall \sigma_- \in \hat{\Sigma}_-.\ \sigma_+\langle \mathsf{sgn}\rangle \neq \sigma_-\langle \mathsf{sgn}\rangle\}$. We want $\widehat{\mathbf{add}}$ to over approximate **add**. If $\hat{\Sigma}_+$ over approximates Σ_+ and $\hat{\Sigma}_-$ is exact, then we can only add states to **add**. There might be some $\sigma_+ \in \hat{\Sigma}_+$ but $\notin \Sigma_+$ in which the state correspondence does not holds for all reachable old states, thus growing $\widehat{\mathbf{add}}$. Now, if $\hat{\Sigma}_+$ is exact, but $\hat{\Sigma}_-$ is an over approximation, we can only remove states from **add**. Now, there may be some $\sigma_- \in \hat{\Sigma}_-$ but $\notin \Sigma_-$ in which some σ_+ has an sgn_+ that equals $\sigma_-\langle \mathsf{sgn}\rangle$. Then, this σ_+ *would not* be in $\widehat{\mathbf{add}}$, but *would* be in **add**. Therefore, to soundly over approximate **add**, we require a representation of old states that *under approximates* all reachable states. Then, we may soundly over approximate **add**.

In order to under approximate reachable states, we use incorrectness logic. Recall the semantics, show below.

$$\models [\varphi]\ s\ [\varphi'] \text{ if and only if}$$
$$\sigma' \models \varphi' \text{ implies } (\sigma \vdash s \Downarrow \sigma' \text{ and } \sigma \models \varphi)$$

All free variables before the "implies" are implicitly universally quantified over, and all free variables after the "implies" are implicitly existentially quantified over. Thus, $\models [\varphi]\ s\ [\varphi']$ indicates that all state satisfying φ' are reachable. This is precisely what we desire.

Now, we can derive the following incorrectness logic triple, $[\top]\ s_-\ [\varphi_-]$ where $\varphi_- = (\mathsf{x}_- > 0 \implies \mathsf{sgn}_- = 1) \wedge (\mathsf{x}_- \leq 0 \implies \mathsf{sgn}_- = -1)$. We cannot soundly add disjunctions or drop conjunctions in incorrectness logic. Then, our formula representing an over approximation of old states that in which the state correspondence does not hold for all new states (under approximate) is $\forall \mathsf{x}_-, \mathsf{sgn}_-.\ \varphi_- \implies \mathsf{sgn}_- \neq \mathsf{sgn}_+$. This implies $\varphi_r := 1 \neq \mathsf{sgn}_+ \wedge -1 \neq \mathsf{sgn}_+$. Then, finally, creating the conjunction with our over approximate φ_+ yields a formula which implies $\mathsf{x}_+ = 0$.

In this section, we saw how partitioning reachable states of the "old" and "new" programs of a revision into the sets **rm**, **rel**, **add** based on a state correspondence is a natural way to represent the semantic effect of program revisions. Then, we noticed that in order to prove safety properties over these sets, we require both an under and over approximating abstract semantics. Section 3 generalizes and formalizes these constructions. Finally, we used Hoare logic and Incorrectness logic to prove a property over **add** and saw how relational Hoare logic was sufficient to prove **rel** safety properties. Section 4 instantiates the parameterized abstract interpreter given in section 3 with the two programs logics.

3 Parameterized Abstract Interpreter

In this section, we build up to a parameterized abstract interpreter for safety properties of removed, related, and added behaviors caused by a program revision. We do not fix an underlying programming language; programs of the underlying language are denoted as s. We write $s_- \pm s_+$ to signify a program s_- that was updated to s_+ by a revision (sometimes referred to as a *biversional* program). If we consider a diff, Δ, to be the changes made by a programmer to s_-, then s_+ may be viewed as the diff applied to the old program ($s_+ = \Delta \circ s_-$). We write σ to represent a behavior of a program s that we will henceforth call *state*. A program state could be a store, trace, or some other notion of program behavior, but we will leave it uninstantiated.

3.1 State Correspondence

As we saw in the section 1, software developers can view a program revision in terms of the behavior (or states) which it removed, added, and kept the same. So, the set of *removed* states (denoted **rm**) is a subset of the set of all possible states of an old program (Σ_-). Thus, any $\sigma_- \in \Sigma_-$ is either $\in$ **rm** or is *related* to some $\sigma_+ \in \Sigma_+$. The set of related states (**rel**) will be tuples of (σ_-, σ_+), and if some $\sigma_- \notin$ **rm**, then there will exist some $(\sigma'_-, \sigma'_+) \in$ **rel** where $\sigma_- = \sigma'_-$. In fact, for all $\sigma_i \in \Sigma_+$ that is related to σ_-, (σ_-, σ_i) is in **rel**. The set of added states (**add**) includes the states that were added by the revision, so states in **add** will not have some $\sigma_- \in \Sigma_-$ that it is related to. Together, the three sets (**rm**, **rel**, **add**) contain all possible program states of both programs and characterize a revision.

Now, we see that to determine if a state is removed, we need an equivalence relation on states, which we will call the *state correspondence* and use $\sim$ to denote it. Given a state correspondence, $\sim$, σ_- is removed if and only if for all $\sigma_+ \in \Sigma_+$, $\sigma_- \not\sim \sigma_+$ (where $\sigma_- \not\sim \sigma_+$ holds if and only if $\sigma_- \sim \sigma_+$ does not hold). This is quite natural, in order to check if some old behavior is removed, you must check it against all new behaviors. Since we will be doing this for-all check quite a bit, we reserve the symbol, $\not\simeq$ for it, shown below.

Definition 1. *The state-lifted state correspondence* ($\not\simeq$)
$\sigma \not\simeq \Sigma'$ *if and only if for all* $\sigma' \in \Sigma', \sigma \nsim \sigma'$
$\sigma \simeq \Sigma'$ *holds if and only if* $\sigma \not\simeq \Sigma'$ *does not hold.*

We write $\simeq$ for the relation, $\not\simeq$, not holding, which can be alternatively defined as: $\sigma \simeq \Sigma'$ if and only if there exists some $\sigma' \in \Sigma'$ where $\sigma \sim \sigma'$.

The state correspondence need not be the same relation when comparing old states to new states versus new states to old states. This is allowed in our framework. $\not\simeq^{\mathbf{rm}}$ indicates that the state correspondence used to to determine if an old state is removed is $\sim^{\mathbf{rm}}$. Likewise for $\not\simeq^{\mathbf{add}}$; if $\sim$ and $\not\simeq$ is used, then it is assumed that $\sim$ is the same for determining added and removed states.

We may now give definitions for **rm**, **rel**, **add**

Definition 2. *The set of removed states (**rm**) based on a set of old states (Σ_-), a set of new states (Σ_+), and a state correspondence ($\not\simeq^{\boldsymbol{rm}}$).*
$\boldsymbol{rm} := \{\sigma_- \in \Sigma_- \mid \sigma_- \not\simeq^{\boldsymbol{rm}} \Sigma_+\}$

Definition 3. *The set of related states (**rel**) based on a set of old states (Σ_-), a set of new states (Σ_+), and state correspondences ($\not\simeq^{\boldsymbol{rm}}$, $\not\simeq^{\boldsymbol{add}}$).*
$\boldsymbol{rel} := \{(\sigma_-, \sigma_+) \in \Sigma_- \times \Sigma_+ \mid \sigma_- \sim^{\boldsymbol{rm}} \sigma_+ \text{ or } \sigma_- \sim^{\boldsymbol{add}} \sigma_+\}$

The set of added states (**add**) is defined as expected. **rel** can be equivalently defined as $\{(\sigma_-, \sigma_+) \in \Sigma_- \times \Sigma_+ \mid \sigma_- \sim^{\mathbf{rm}} \sigma_+\} \cup \{(\sigma_-, \sigma_+) \in \Sigma_- \times \Sigma_+ \mid \sigma_- \sim^{\mathbf{add}} \sigma_+\}$. The left side represents all $\sigma_- \notin \mathbf{rm}$ paired with the new states they are related to. The right side represents all $\sigma_+ \notin \mathbf{add}$ which are related to some σ_- paired with its related old states. Now, we can see how this framework generalizes the concrete semantics of "all reachable pairs" frameworks like relational Hoare logic. If we let $\sim$ hold universally, then **rel** is the set of all reachable pairs.

3.2 Concrete Semantics

Now we have a way to partition reachable states into the representation we are trying to prove safety properties over. However, we have not internalized the computation of reachable states; thus, we do not have a "machine" to abstract. If we had an abstraction or program logic that was expressive enough to soundly encode an over-approximation of **rm**, **rel**, **add**, we could use that and would not need to build the partition into the concrete semantics. However, we feel that the exercise of building the partition into the semantics is necessary to illuminate the intricacies of over-approximating these sets.

We will define our concrete semantics with the following judgment form: $\Sigma_-, \Sigma_+ \vdash s_- \pm s_+ \Downarrow \mathbf{rm}, \mathbf{rel}, \mathbf{add}$. Its "input", or context, is sets of initial new and old states. Then, it computes our three sets, **rm**, **rel**, **add**, which can be seen as the "value" type. It will also be dependent on state correspondences. This looks more like a collecting semantics than concrete, so it may be tempting to define something like $\sigma_-, \sigma_+ \vdash s_- \pm s_+ \Downarrow (\sigma'_-, (\sigma''_-, \sigma''_+), \sigma'_+)$. Then, define a collecting semantics over this to lift each component to states and abstract those states. However, recall from section 2.1 that state comparisons do not give us enough information to determine if a reachable old state is in **rm** or **rel**; we require sets. Therefore, after stepping a σ_- to σ'_-, we would not be able to tell where to put it in right side of our concrete semantics relation.

It also may be tempting to define both sides as Σ_-, Σ_+, since that would give us sufficient information to compute **rm**, **rel**, **add**. But remember, we are giving the semantics of our analysis that is an over-approximation of **rm**, **rel**, **add**, so we want our semantics to compute **rm**, **rel**, **add**.

Otherwise, we would need to define a projection from to Σ_-, Σ_+ to **rm**, **rel**, **add**, but this confuses and complicates our abstraction and soundness condition. Furthermore, **rm**, **rel**, **add** on the left side would be unnatural since with no programs, there is no added or removed states. This restricts the compositionality of the semantics, but a version with the **rm**, **rel**, **add** triple on both sides was investigated, and still did not allow for compositionality. This leads us to our concrete semantics.

Definition 4. *The concrete semantics for our parameterized abstract interpreter. Parameterized by state correspondences ($\sim^{\boldsymbol{rm}}, \sim^{\boldsymbol{add}}$), an underlying programming language, a state representation of it, and it's semantics ($\sigma \vdash s \Downarrow^* \sigma'$). We take the natural collecting lift of these semantics ($\Sigma \vdash s \Downarrow^{\{*\}} \Sigma'$).*

$$\frac{\begin{array}{c}\Sigma_- \vdash s_- \Downarrow^{\{*\}} \Sigma'_- \quad \Sigma_+ \vdash s_+ \Downarrow^{\{*\}} \Sigma'_+ \\ \boldsymbol{rm} = \{\sigma'_- \in \Sigma'_- \mid \sigma'_- \not\sim^{\boldsymbol{rm}} \Sigma'_+\} \\ \boldsymbol{add} = \{\sigma'_+ \in \Sigma'_+ \mid \sigma'_+ \not\sim^{\boldsymbol{add}} \Sigma'_-\} \\ \boldsymbol{rel} = \{(\sigma'_-, \sigma'_+) \in \Sigma'_- \times \Sigma'_+ \mid \sigma'_- \sim^{\boldsymbol{rm}} \sigma'_+ \text{ or } \sigma'_- \sim^{\boldsymbol{add}} \sigma'_+\}\end{array}}{\Sigma_-, \Sigma_+ \vdash s_- \pm s_+ \Downarrow \boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add}}$$

We see that to evaluate from Σ_-, Σ_+ to **rm**, **rel**, **add** based on a biversional program, we step the initial states then compute the the sets via the state correspondence. Now, we can see that **rm**, **rel**, **add** is a particular view of the biversional state space which requires the whole space in order to be computed. Notice that we are assuming that base programming language is imperative, meaning that it's step relation is one in which the context type is the same as the value type. Allowing a underlying program semantics where the context type does not equal the value type is possible and fairly straightforward, but left as future work.

3.3 Collecting Semantics

Given our concrete semantics, we may now define a collecting semantics and abstract domains. First, let's take a slight detour to discuss the abstract interpretation recipe we will use.

Definition 5. *A general abstract interpretation framework*

(1) *Concrete semantics:* $\sigma \vdash s \Downarrow v$
(2) *Collecting semantics:* $\Sigma \vdash s \Downarrow^{\{\}} V$
(3) *Abstract semantics:* $\hat{\sigma} \vdash s \Downarrow^{\sharp} \hat{v}$

Where $\Sigma : \mathcal{P}(\sigma)$, $V : \mathcal{P}(v)$ *and* $\gamma(\hat{\sigma}) = \{\sigma \mid \sigma \models \hat{\sigma}\}$, $\gamma(\hat{v}) = \{v \mid v \models \hat{v}\}$. *Often, for imperative languages,* $\sigma = v$.

In order to define an abstract semantics, we first define a concrete and collecting semantics. The essence of this recipe is that our collecting semantics defines the best possible abstract interpreter. That is, the soundness of the abstraction is defined w.r.t. the collecting semantics. Furthermore, if we had a sound and complete abstract interpreter, it would simply be the collecting semantics. Thus, the collecting semantics could lose precision from simply collecting all possible concrete states. Another important aspect of this recipe is that the collected state, Σ, is a collection of σ, and our abstract state, $\hat{\sigma}$, concretizes to a collected state. This allows for γ to be defined as all σ that models a $\hat{\sigma}$.

Given the semantics outlined in definition 5, we can define soundness of a typical over-approximating abstract interpreter as follows.

Definition 6. *Soundness of an over-approximating abstract interpretation framework defined with definition 5.*
$\hat{\sigma} \vdash s \Downarrow^{\sharp} \hat{v}$ *and* $\gamma(\hat{\sigma}) \vdash s \Downarrow^{\{\}} V$ *implies* $V \subseteq \gamma(\hat{v})$.
We will write $\Downarrow^{\uparrow}$ *for an over-approximating abstract semantics.*

Our recipe does not enforce the abstract interpreter to be over-approximating. Below, we define the soundness of an abstraction that under-approximates the collecting semantics for bug-finding or incorrectness reasoning.

Definition 7. *Soundness of an under-approximating abstract interpretation framework defined with definition 5.*
$\hat{\sigma} \vdash s \Downarrow^{\sharp} \hat{v}$ *and* $\gamma(\hat{\sigma}) \vdash s \Downarrow^{\{\}} V$ *implies* $\gamma(\hat{v}) \subseteq V$.
We will write $\Downarrow^{\downarrow}$ *for an under-approximating abstract semantics.*

Now, we can follow this recipe to define the collecting semantics. Our collected values, V, must be collection of v which is **rm**, **rel**, **add**. The same holds for our collected states. In order to lose no precision, we will define the typical set-lifted semantics.

Definition 8. *The collecting semantics of our parameterized abstract interpreter.* $\sigma \vdash s_- \pm s_+ \Downarrow v$ *is the concrete semantics of our abstract interpreter given in definition 4.*
$\Sigma ::= \circ \mid \Sigma, (\boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add})$

$$\frac{V = \{v \mid \sigma \in \Sigma \text{ and } \sigma \vdash s_- \pm s_+ \Downarrow v\}}{\Sigma \vdash s_- \pm s_+ \Downarrow^{\{\}} V}$$

Our collecting semantics collect *sets* of **rm**, **rel**, **add** triples, so our abstract value, $\hat{v}$, must concretize to a set of **rm**, **rel**, **add** triples. However, each biversional program and state correspondence will produce a singular **rm**, **rel**, **add**, and this triple is what safety properties will be checked against. Thus, it is natural for $\hat{v}$ to be an over-approximation of the three sets. Then, its concretization would be a **rm**, **rel**, **add** triple.

3.4 First Abstraction

Since this set-of-sets is not what we want, our first abstraction will be taking the set-of-sets back down to a set (*i.e.* lowering our collecting semantics to concrete semantics).

Definition 9. *Our first abstraction.* $\hat{\sigma} := (\Sigma_-, \Sigma_+), \hat{v} := (\boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add})$.
Abstract evaluation

$$\frac{(\Sigma_-, \Sigma_+) \vdash s_- \pm s_+ \Downarrow \boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add}}{(\Sigma_-, \Sigma_+) \vdash s_- \pm s_+ \Downarrow^{\sharp} \boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add}}$$

Value abstraction function.

$$\begin{aligned}
&\alpha : \mathcal{P}(V) \to V \\
&\alpha(V, (\boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add})) := (\boldsymbol{rm} \cup \alpha(V)_1, \boldsymbol{rel} \cup \alpha(V)_2, \boldsymbol{add} \cup \alpha(V)_3) \\
&\alpha(\circ) := (\{\}, \{\}, \{\})
\end{aligned}$$

State abstraction function.

$$\begin{aligned}
&\alpha : \mathcal{P}(\Sigma) \to \Sigma \\
&\alpha(\Sigma, (\Sigma_-, \Sigma_+)) := (\boldsymbol{rm} \cup \alpha(\Sigma)_1, \boldsymbol{rel} \cup \alpha(\Sigma)_2) \\
&\alpha(\circ) := (\{\}, \{\}, \{\})
\end{aligned}$$

Element-wise state abstraction function (β) is the identity function.

Value concretization function.

$$\begin{gathered}\boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add} \models \boldsymbol{rm}', \boldsymbol{rel}', \boldsymbol{add}' \textit{ if and only if} \\ \boldsymbol{rm} \subseteq \boldsymbol{rm}' \textit{ and } \boldsymbol{rel} \subseteq \boldsymbol{rel}' \textit{ and } \boldsymbol{add} \subseteq \boldsymbol{add}' \\ \gamma(\hat{v}) := \{v \mid v \models \hat{v}\}\end{gathered}$$

State concretization function.

$$\begin{gathered}\Sigma_-, \Sigma_+ \models \Sigma'_-, \Sigma'_+ \textit{ if and ony if} \\ \Sigma_- \subseteq \Sigma_- \textit{ and } \Sigma_+ \subseteq \Sigma_+ \\ \gamma(\hat{\sigma}) := \{\sigma \mid \sigma \models \hat{\sigma}\}\end{gathered}$$

Now, we will show that in many cases, we can use a much more precise value concretization function that makes this abstraction sound and complete. First, notice that "running" the abstract interpreter, will start by abstracting a single concrete state, $\beta(\sigma)$ (in this case, $\sigma = \Sigma_-, \Sigma_+$). In all of these cases, we can refine our abstraction to be sound and complete.

Our concrete semantics form a total and well-defined function: $\lambda(\sigma, s_- \pm s_+).v$ such that $\sigma \vdash s_- \pm s_+ \Downarrow v$ where $\Downarrow$ is the concrete evaluation from definition 4. Thus, for any $\sigma = \Sigma_-, \Sigma_+$, $\{\sigma\} \vdash s_- \pm s_+ \Downarrow^{\{\}} V$ where $\Downarrow^{\{\}}$ is given in definition 8, V will have cardinality one. That is, $V = \{(\mathbf{rm}, \mathbf{rel}, \mathbf{add})\}$ where $\mathbf{rm}, \mathbf{rel}, \mathbf{add}$ is the value produced by concrete semantics for σ. This implies that $\beta(\sigma) \vdash s_- \pm s_+ \Downarrow^\sharp \hat{v}$ will produce a $\hat{v}$ which is exactly all possible concrete values. Then, we can use alter the value concretization function for the abstraction given in 9 to achieve a sound and complete abstraction.

Definition 10. *An alteration to definition 9 that achieves a sound and complete abstraction. The value concretization function is redefined as follows in the case when $\hat{\sigma} = \beta(\sigma)$.*

$$\begin{gathered}\boldsymbol{rm}, \boldsymbol{rel}, \boldsymbol{add} \models \boldsymbol{rm}', \boldsymbol{rel}', \boldsymbol{add}' \textit{ if and only if} \\ \boldsymbol{rm} = \boldsymbol{rm}' \textit{ and } \boldsymbol{rel} = \boldsymbol{rel}' \textit{ and } \boldsymbol{add} = \boldsymbol{add}' \\ \gamma(\hat{v}) := \{v \mid v \models \hat{v}\}\end{gathered}$$

Anytime $\beta(\sigma)$ is used as the context for $\Downarrow^\sharp$, this redefined $\Downarrow^\sharp$ is used.

Regardless of the existence of the alteration to γ presented in definition 10, after this initial abstraction, we are left we the task of abstracting $\hat{\sigma}, \hat{v}$ which is the same as abstracting σ, v. Furthermore, with the special case of γ when $\hat{\sigma} = \beta(\sigma)$, we can ensure that our initial abstraction loses no precision. Thus, completeness can be determined w.r.t. any additional abstractions of $\sigma = (\Sigma_-, \Sigma_+)$ and $v = (\mathbf{rm}, \mathbf{rel}, \mathbf{add})$.

3.5 Parameterized Abstract Semantics

Here, we give a parameterized abstract interpreter which can be instantiated with different abstract domains for the underlying program state space, Σ. As we saw in section 3.4, we are left with the task of abstracting our concrete semantics defined in definition 4. Viewing our entire landscape of semantics, this is actually an abstraction of the abstraction given in definition 9.

From section 2.2, recall that in order to soundly over approximate **add**, we need an over approximating abstraction of reachable new states with an under approximating abstraction of reachable old states. We will write, $\Downarrow^\uparrow$ for an over approximating abstract semantics in a domain $\mathcal{D}$ (and $\Downarrow^\downarrow$ for under). We let elements in $\mathcal{D}$ be denoted as $\hat{\sigma}$. Abstract state elements, $\hat{\sigma}$, concretize to a set of states. Therefore, to abstract our concrete semantics context, Σ_-, Σ_+, we will have a pair of abstract state elements: $\hat{\sigma}_-, \hat{\sigma}_+$. These elements are in the same domain that will be used in the parameterized abstract semantics for the underlying programs. Furthermore, the abstraction of **rm** and **add** will also be in this domain since they must concretize to a set of states. Our abstraction of **rel** should concretize to a set of pairs of states, so it will be in the domain $\mathcal{D} \times \mathcal{D}$.

Now, we see our parameterized abstract interpreter is parameterized by a single domain, $\mathcal{D}$. This is akin to the fact that our concrete semantics (definition 4) was also parameterized by a state representation of the underlying programs. The only aspect of our concrete semantics left to abstract is the state correspondence check, which has some intricacies discussed below. We found that the choice of how to abstract this check is dependent on the domain $\mathcal{D}$; so in our abstract interpreter, we leave the this check as a required operation on the domain.

Definition 11. *Parameterized abstract semantics. Parameters are a programming language whose programs are denoted s, a domain $\mathcal{D}$ whose elements are abstractions of concrete states of s, and an over and under approximating abstract semantics on $\mathcal{D}$ an s. Furthermore, $\mathcal{D}$ must be a lattice that implements $\hat{\not\approx}$. In doing that, $\mathcal{D}$ will also implement $\hat{\sim}$.*

$$\frac{\hat{\sigma}_- \vdash s_- \Downarrow^{\uparrow} \hat{\sigma}'_- \quad \hat{\sigma}_+ \vdash s_+ \Downarrow^{\uparrow} \hat{\sigma}'_+ \quad \hat{\sigma}_- \vdash s_- \Downarrow^{\downarrow} \hat{\sigma}''_- \quad \hat{\sigma}_+ \vdash s_+ \Downarrow^{\downarrow} \hat{\sigma}''_+ \\ \hat{\sigma}'_- \,\hat{\not\approx}^{\boldsymbol{rm}}\, \hat{\sigma}''_+ \sqsubseteq \hat{\boldsymbol{rm}} \quad \hat{\sigma}'_+ \,\hat{\not\approx}^{\boldsymbol{add}}\, \hat{\sigma}''_- \sqsubseteq \hat{\boldsymbol{add}} \quad [(\hat{\sigma}'_- \,\hat{\sim}^{\boldsymbol{rm}}\, \hat{\sigma}'_+) \sqcup (\hat{\sigma}'_- \,\hat{\sim}^{\boldsymbol{add}}\, \hat{\sigma}'_+)] \sqsubseteq \hat{\boldsymbol{rel}}}{\hat{\sigma}_-, \hat{\sigma}_+ \vdash s_- \pm s_+ \Downarrow^{\sharp} \hat{\boldsymbol{rm}}, \hat{\boldsymbol{add}}, \hat{\boldsymbol{rel}}}$$

First, we notice the use of $\sqsubseteq$. We want to prove safety properties over, say **rm**, so any $\hat{\mathbf{rm}}$ that concretizes to a set of states that contains the "computed" $\hat{\mathbf{rm}}' = \hat{\sigma}'_- \,\hat{\not\approx}^{\mathbf{rm}}\, \hat{\sigma}''_+$, is sound. This has the practical effect of allowing weakening within the abstract interpretation.

Now, we turn our attention to discussing $\hat{\not\approx}$. The idea is that this abstracts removed or added states. Recall the definition of removed states from our concrete semantics, $\mathbf{rm} = \{\sigma'_- \in \Sigma'_- \mid \sigma'_- \not\approx^{\mathbf{rm}} \Sigma'_+\}$, and unrolling $\not\approx$ (definition 1) gives us $\mathbf{rm} := \{\sigma_- \in \Sigma_- \mid \forall \sigma_+ \in \Sigma_+ .\ \not\sim \sigma_+\}$.

The soundness of $\hat{\not\approx}$ is defined as follows.

$$\{\sigma \in \gamma(\hat{\sigma}) \mid \sigma \not\approx \gamma(\hat{\sigma}')\} \subseteq \gamma(\sigma \,\hat{\not\approx}\, \hat{\sigma}')$$

The glaring problem with such an abstraction is that, usually, there is no way to "iterate" through abstract elements. We can naturally lift $\sim$ to our abstract domain, $\hat{\sim}$ (and in turn $\hat{\not\sim}$), by defining it on our abstract elements instead of concrete states. However, if we have no way to "iterate" through $\hat{\sigma}$, then we must characterize an *entire* $\hat{\sigma}_-$ as either removed or not removed. This simple abstraction of removed states defines $\hat{\sigma} \,\hat{\not\approx}\, \hat{\sigma}'$ as $\hat{\sigma} \,\hat{\not\sim}\, \hat{\sigma}'$. We can see this as the concrete definition of **rm** with only one element in Σ_- and Σ_+.

This is quite a coarse abstraction of removed and added states (all or none). Furthermore, we notice that our **rm** property is interpreted over old states, but requires the entire set of new states to compute it. However, this simple **rm** abstraction actually reduces **rm** properties to no longer require the set of new states. Instead, it can be formulated as a property dependent only on state pairs. Since with this abstraction either all states will be in **rm** or none, it is equivalent to checking $\sigma \sim \sigma'$ or $\sigma \not\sim \sigma'$ over reachable pairs. These types of properties can be proven in many existing frameworks, so we note that the additional properties required of $\mathcal{D}$ are essential to our approach.

Now, we present two ways to implement $\hat{\not\approx}$. Option 1 places a larger restriction on $\mathcal{D}$, while option 2 is more permissive but most natural for symbolic domains. While 2 could probably be implemented for most domains, it will likely require an additional abstraction to some symbolic domain. The instantiation of our parameterized abstract interpreter presented in section 4 will use option 2 with a symbolic domain.

(1) We can achieve a more precise $\hat{\not\approx}$, that cannot be reduced to a predicate over pairs of states, if $\mathcal{D}$ supports finite partitioning. A finitely partitioned abstract domain is such that $\hat{\sigma} = \hat{\sigma}_1, \hat{\sigma}_2, \ldots, \hat{\sigma}_n$ where $\gamma(\hat{\sigma}_i) = \Sigma_i$ and $\gamma(\hat{\sigma}) = \gamma(\hat{\sigma}_1) \cup \gamma(\hat{\sigma}_2) \cup \ldots \cup \gamma(\hat{\sigma}_n)$. Then, we may

define $\hat{\sigma} \mathrel{\hat{\not\approx}} \hat{\sigma}'$ as $\forall \hat{\sigma}_i \in \hat{\sigma}.\ (\forall \hat{\sigma}'_i \in \hat{\sigma}'.\ \hat{\sigma}_i \mathrel{\hat{\not\sim}} \hat{\sigma}'_i)$ where $\hat{\not\sim}$ is the $\not\sim$ relation lifted to our abstract domain (defining the operator on the new type). Such a domain exists.

(2) For domains which are not finitely partitioned, hope is not lost. Let's consider abstracting removed states with $\hat{\not\approx}^{\mathbf{rm}}$. If we had an abstract element, say $\hat{\sigma}'_-$ that represented all possible states where for any one of those states, say $\sigma'_- \in \gamma(\hat{\sigma}'_-)$, $\forall \sigma_+ \in \gamma(\hat{\sigma}_+).\ \sigma'_- \not\sim^{\mathbf{rm}} \sigma_+$ held; then we could meet that with the reachable state element to achieve an abstraction of **rm**, $\hat{\sigma}_- \sqcap \hat{\sigma}'_-$. This meet would represent all reachable states that also satisfy our set lifted state correspondence relation (*i.e.* removed condition).
Now, we are left with constructing a $\hat{\sigma}'_-$ such that $\forall \sigma'_- \in \gamma(\hat{\sigma}'_-).\ (\forall \sigma_+ \in \gamma(\hat{\sigma}_+).\ \sigma'_- \not\sim^{\mathbf{rm}} \sigma_+)$. First, we make the following observation. $\hat{\sigma}$ is a state predicate, we even defined its concretization in terms of all concrete states which model it. We can lift this state predicate, say $\hat{\sigma}_+$, to a global property by closing it with respect to variables modified by s_+. If we, say, universally quantify over these modified variables, then we achieve a property (that is not a state predicate) which holds for all possible valuations of the modified variable, *i.e.* a property that represents all states which $\hat{\sigma}_+$ holds.
A representation of all pairs of states which satisfies $\not\sim$, say $\hat{\sigma}_{sc}$, is a predicate on pairs of states. We could do the same trick to remove the dependency on one of the states, lowering it to a single state predicate. Now, we have the tools we need to construct our $\hat{\sigma}'_-$. We let η_+ be a function which given a predicate on *at least* σ_+, say φ, removes the dependency on old states by producing a new assertion that holds for all σ_+ which φ represented.

Definition 12. *An abstraction of the removed state operator ($\hat{\not\approx}^{rm}$) for domains that are not finitely partitioned. We require $\hat{\sigma}_{sc}$ and η_+ such that:*

$$\{(\sigma_-, \sigma_+) \mid \sigma_- \not\sim^{rm} \sigma_+\} \subseteq \gamma(\hat{\sigma}_{sc})$$
$$\{\sigma_- \mid \forall \sigma_+ \in \gamma(\hat{\sigma}_+).\ (\sigma_-, \sigma_+) \models \hat{\sigma}_{sc}\} \subseteq \gamma(\eta_+(\hat{\sigma}_+ \sqsubseteq \hat{\sigma}_{sc}))$$

$\eta_+(\hat{\sigma}_+ \sqsubseteq \hat{\sigma}_{sc})$ can equivalently be defined as all σ_- such that $\gamma(\hat{\sigma}_+) \subseteq \gamma(\hat{\sigma}_{sc})$ and $\sigma_- \in \{\sigma'_- \mid (\sigma'_+, \sigma'_-) \in \gamma(\hat{\sigma}_{sc})\}$ holds.
In order to define $\hat{\sigma}_{sc}$, a definition $\hat{\sim}$ will be required. This will be used to compute $\hat{\mathbf{rel}}$. We require the same constructions for $\hat{\not\approx}^{add}$.
Now, we have $\hat{\sigma}_- \hat{\not\approx}^{rm} \hat{\sigma}_+$ if and only if $\hat{\sigma}_- \sqcap \eta_+(\hat{\sigma}_+ \sqsubseteq \hat{\sigma}_{sc})$

Definition 13. *An instantiation of η from definition 12 for symbolic domains which include universal quantification in the assertion language.*

$$\eta_+(\hat{\sigma}_2) := \forall mod(s_+).\ \hat{\sigma}_2$$

Where $mod(s_+)$ is all variables modified by s_+.

4 Instantiation with Variable Correspondence, Hoare Logic, Incorrectness Logic, and IMP

Now, we will instantiate the parameterized abstract interpreter given in section 3.5.

Our underlying programming language will be IMP with non-deterministic assignment whose syntax is given in section A. The concrete semantics of imp is given with the judgment form $\sigma \vdash s \Downarrow^* \sigma'$. σ is a map from program variables to concrete values, which are either numbers or booleans.

Now, we will define an instantiation of state correspondence for IMP, which we will call the variable correspondence. The grammar and semantics are given below.

Definition 14. *An instantiation of state correspondence for IMP, which we will refer to as the* variable correspondence.

$$\begin{array}{rrcl} \textit{variable correspondence} & vc & ::= & (u)\,,(u') \\ \textit{universional state bit} & u & ::= & u\,,x \mid x' \\ \textit{variables} & x & & \end{array}$$

With the variable correspondence, $\sim^{\mathbf{rm}} = \sim^{\mathbf{add}}$. Furthermore, a vc, say u, u' is well formed only if u has the same size as u'. However, u need not have the same number of unique variables as u'.

For a given $vc := (x_1, \ldots x_n), (x'_1, \ldots x'_n)$, we define $\sim$ as follows.

$$\sigma_- \sim \sigma_+ \textit{ if and only if } (\sigma_-\langle x_1\rangle, \ldots, \sigma_-\langle x_n\rangle) \neq (\sigma_+\langle x'_1\rangle, \ldots, \sigma_+\langle x'_n\rangle)$$

Where $\sigma\langle x\rangle$ looks up the variable x in state σ.

Now, we will instantiate our over approximating abstract semantics with Hoare logic.

$\hat{\sigma} \vdash s \Downarrow^{\uparrow} \hat{\sigma}' := \vdash \{\hat{\sigma}\}\ s\ \{\hat{\sigma}'\}$

And our under approximating abstract semantics with Incorrectness logic.

$\hat{\sigma} \vdash s \Downarrow^{\downarrow} \hat{\sigma}' := \vdash [\hat{\sigma}]\ s\ [\hat{\sigma}']$

$\hat{\sigma}$ will be formulae in an assertion language which is first order logic along with IMP expression. $\hat{\sigma}_-$ will include variables subscripted with $-$, and likewise for $\hat{\sigma}_+$ and $+$. Abstract elements which represent state pair predicates may contain variables with both $-$ and $+$ subscripts.

Now, all that is left to define is $\hat{\not\approx}$. First, for the $vc := (x^1, \ldots, x^n), (y^1, \ldots, y^n)$ we define $\hat{\sigma}_{sc}$ as $(x^1_-, \ldots x^n_-) \neq (y^1_+, \ldots, y^n_+)$, and $\hat{\sigma}'_{sc}$ as $(x^1_-, \ldots x^n_-) = (y^1_+, \ldots, y^n_+)$ We define $\hat{\sigma}\ \hat{\not\approx}\ \hat{\sigma}'$ as $\hat{\sigma} \wedge (\forall mod(s).\ \hat{\sigma}' \implies \hat{\sigma}_{sc})$ when $\hat{\sigma}'$ represents reachable states of program s. This leaves us with an instantiated abstract interpreter.

Definition 15. *Our abstract interpreter with IMP, vc, Hoare logic, and Incorrectness logic.*

$$\frac{\begin{array}{c} \vdash \{\hat{\sigma}_-\}\ s_-\ \{\hat{\sigma}'_-\} \quad \vdash \{\hat{\sigma}_+\}\ s_+\ \{\hat{\sigma}'_+\} \quad \vdash [\hat{\sigma}_-]\ s_-\ [\hat{\sigma}''_-] \quad \vdash [\hat{\sigma}_+]\ s_+\ [\hat{\sigma}''_+] \quad (\hat{\sigma}'_- \wedge \hat{\sigma}'_+ \wedge \hat{\sigma}'_{sc}) \implies \hat{\mathbf{rel}} \\ [\hat{\sigma}'_- \sqcap (\forall mod(s_+).\ \hat{\sigma}''_+ \implies \hat{\sigma}_{sc})] \implies \hat{\mathbf{rm}} \quad [\hat{\sigma}'_+ \sqcap (\forall mod(s_-).\ \hat{\sigma}''_- \implies \hat{\sigma}_{sc})] \implies \hat{\mathbf{add}} \end{array}}{\hat{\sigma}_-, \hat{\sigma}_+ \vdash s_- \pm s_+ \Downarrow^{\sharp} \hat{\mathbf{rm}}, \hat{\mathbf{add}}, \hat{\mathbf{rel}}}$$

The major difference from our parameterized abstract interpreter is the computation of $\hat{\mathbf{rel}}$. We had $[(\hat{\sigma}'_-\ \hat{\sim}^{\mathbf{rm}}\ \hat{\sigma}'_+) \sqcup (\hat{\sigma}'_-\ \hat{\sim}^{\mathbf{add}}\ \hat{\sigma}'_+)] \sqsubseteq \hat{\mathbf{rel}}$. Since $\sim^{\mathbf{rm}} = \sim^{\mathbf{add}}$ and $\sqsubseteq$ is $\implies$ in our domain, we first simplify to $(\hat{\sigma}'_-\ \hat{\sim}\ \hat{\sigma}'_+) \implies \hat{\mathbf{rel}}$. Now, the soundness of $\hat{\sim}$ is defined as $\{(\sigma'_-, \sigma'_+) \in \gamma(\hat{\sigma}'_-) \times \gamma(\hat{\sigma}'_+) \mid \sigma'_- \sim \sigma'_+\} \subseteq \gamma(\hat{\sigma}'_-\ \hat{\sim}\ \hat{\sigma}'_+)$. We notice that $\gamma(\hat{\sigma}'_- \wedge \hat{\sigma}'_+) = \{\gamma(\hat{\sigma}'_-) \times \gamma(\hat{\sigma}'_+)\}$ and $\gamma(\hat{\sigma}'_{sc}) = \{(\sigma_-, \sigma_+) \mid \sigma_- \sim \sigma_+\}$. Thus, $\gamma(\hat{\sigma}'_- \wedge \hat{\sigma}'_+ \wedge \hat{\sigma}'_{sc}) = \{(\sigma'_-, \sigma'_+) \in \gamma(\hat{\sigma}'_-) \times \gamma(\hat{\sigma}'_+) \mid \sigma'_- \sim \sigma'_+\}$.

5 Evaluation

In this section, we use the abstract interpreter defined in definition 15 to prove properties about IMP program revisions. We will use very simple examples that illustrate the coherence of our constructions with respect to program difference properties.

First, let's consider `x := 1` $\pm$ `y := 2`. With $vc := (x), (y)$, we see that all old states $(x_- = 1)$ are removed, and all new states $(y_+ = 2)$ are added. No reachable states satisfy the state correspondence $(x_- = y_+)$, so **rel** is empty. This is all quite intuitive, but what about about $vc := (x), (x)$? Let's check if $\hat{\mathbf{add}} := (y_+ = 2)$ is derivable. We have $\vdash [\top]$ `x := 1` $[x_- = 1]$ and $\vdash \{\top\}$ `y := 2` $\{y_+ = 2\}$. We generate the verification condition: $[y_+ = 2 \wedge (\forall x_-. x_- = 1 \implies x_- \neq x_+)] \implies y_+ = 2$. This is certainly satisfiable, so $y_+ = 2$ is a sound over approximation of **add**. However, the set of reachable new states will always be a sound over approximation of **add**, which we can clearly see with the formula above.

Let's see if we can pin down **add** any better. Well, the antecedent of our verification condition precisely describes **add**. It reduces to $y_+ = 2 \land x_+ \neq 1$. This makes sense, as it will be true of all added states.

6 Related Work

Verifying properties of the difference between two programs is not new. Relational Hoare Logic (RHL) [3] defines a means to reason about the relationship between two programs. It is used to prove many categories of properties which frame and formalize the semantics of a code change such as regression verification [4], translation validation [7], contextual equivalence [8], refinement, and others [2]. The strategy used to prove these properties is often either the construction of a product program [1] to reduce the problem to a functional safety property, or to rely on similarities between the two programs.

Some work has been done to develop analysis tools and techniques particularly for program diffs and their properties of interest. Semantic Diff [5] and differential symbolic execution [10] are tools which summarize the behavioral differences between two version of a program. Differential assertion checking [6] checks if assertion which holds in one program does or does not hold in another. There is also work [re-find citation] using sound abstract interpretation techniques to prove output equivalence given the same inputs.

The related work which most closely resembles ours is "Abstract Semantic Differencing for Numerical Programs" [9]. It presents an abstract interpretation framework that over approximates program differences. This final abstract state of this tool is a list of abstracted guard conditions and the abstract values of correlated variables under certain guard conditions. This can be seen as summarizing the differences, providing bounds for those differences, and providing conditions for the program variables to be equal. This work also has the concept of added, removed, and related states; however, these terms are used to mean a partition on the final abstract state. So, for example, a new value for a certain variable will be in the added state. This work also relies on a product program, variable correspondence, and correlating abstract domain.

7 Conclusion

We introduced a concrete semantics that describes the semantic effect of program revisions which characterizes reachable states as removed, related, or added. These sets are created via a parameterized equivalence relation on states which ensures the semantics are grounded in the intuitive notion developers have about program differences. We saw the intricacies associated with proving (safety) program difference properties which necessitated precise abstract domains, and both over and under approximating abstract semantics. A heavily parameterized abstract interpreter for program revision properties was introduced, then an instantiation of it with Hoare Logic and Incorrectness Logic was used to prove some simple examples.

Acknowledgments

This report is based on a project advised by Bor-Yuh Evan Chang with help from Matthew Hammer. Of course, this project would not exist without them. The idea originated from a grant submitted by Evan Chang, and their insights, guidance, and contributions have been invaluable in the creation of this paper. This paper is not solely authored by me, but since it is a final project, I will instead acknowledge that here.

A IMP

Definition 16. *IMP grammar.*

$$
\begin{array}{rccl}
\text{expressions} & e & ::= & n \mid \texttt{x} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \div e_2 \mid e_1 \,\%\, e_2 \\
\text{boolean expressions} & b & ::= & b^v \mid e_1 = e_2 \mid e_1 \texttt{ != } e_2 \mid b_1 \,\&\&\, b_2 \mid b_1 \texttt{ || } b_2 \mid e_1 \texttt{ >= } e_2 \mid e_1 < e_1 \mid \\
\text{values} & v & ::= & n \\
\text{commands} & s & ::= & \texttt{x := } e \mid \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \mid \\
 & & & s_1 \texttt{ ; } s_2 \mid \texttt{skip} \mid \texttt{while(}b\texttt{)\{}s\texttt{\}} \mid \texttt{x := !} \\
\text{variables} & \texttt{x} & & \\
\text{numbers} & n & & \\
\text{booleans} & b^v & &
\end{array}
$$

References

[1] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods*, Michael Butler and Wolfram Schulte (Eds.). Springer, Berlin, Heidelberg, 200–214. doi:10.1007/978-3-642-21437-0_17

[2] Bernhard Beckert and Mattias Ulbrich. 2018. Trends in Relational Program Verification. In *Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, Peter Müller and Ina Schaefer (Eds.). Springer International Publishing, Cham, 41–58. doi:10.1007/978-3-319-98047-8_3

[3] Nick Benton. [n. d.]. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. ([n. d.]).

[4] Benny Godlin and Ofer Strichman. [n. d.]. Regression Verification: Proving the Equivalence of Similar Programs. ([n. d.]).

[5] Jackson and Ladd. 1994. Semantic Diff: a tool for summarizing the effects of modifications. In *Proceedings 1994 International Conference on Software Maintenance*. 243–252. doi:10.1109/ICSM.1994.336770

[6] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, Saint Petersburg Russia, 345–355. doi:10.1145/2491411.2491452

[7] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Portland OR USA, 22–32. doi:10.1145/2737924.2737965

[8] Andrzej S Murawski, Steven J Ramsay, and Nikos Tzevelekos. [n. d.]. Game Semantic Analysis of Equivalence in IMJ. ([n. d.]).

[9] Nimrod Partush and Eran Yahav. 2013. Abstract Semantic Differencing for Numerical Programs. In *Static Analysis*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Francesco Logozzo, and Manuel Fähndrich (Eds.). Vol. 7935. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–258. doi:10.1007/978-3-642-38856-9_14 Series Title: Lecture Notes in Computer Science.

[10] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 226–237. doi:10.1145/1453101.1453131