2 Developer-defined Relations over Objects

WWhen a developer programs to a framework, the primary task is not about creating new objects or data. In many cases, programming in this environment is about *manipulating the abstract associations between existing objects*. Every time the plugin receives a callback from the framework, it is implicitly notified of the current associations between objects. As the plugin calls framework methods, the framework changes these associations, and the plugin learns more about how the objects relate. Every method call, field access, or test gives the plugin more information. Even when the plugin needs to create a new object, it is frequently done by calling abstract factory methods that set up the object and its relationships with other objects.

The ASP.NET framework exemplifies this means of interaction. In the `DropDownList` example, all the objects are provided by the framework, and the plugin simply changes their relationships with each other through calls to the framework. In fact, the `DropDownList` itself, and the data within it, is frequently set up using dependency injection, a mechanism in which the framework populates the fields of the plugin based on an external configuration file [7]. This may be done in several stages, with the framework notifying the plugin as it completes each stage using a callback. When using dependency injection, the plugin simply receives and manipulates preconfigured objects.

Since the primary mechanism of interaction is based on manipulating relationships between objects, we will model it formally using a mathematical relation. A *relation* is a named, mathematical relation on several types τ.

$$\texttt{Relation} ::= \texttt{name} \rightarrow \tau_1 \times \ldots \times \tau_n$$

A *relationship* is a single tuple in a relation, represented as

$$\texttt{Relationship} ::= \texttt{name}(\ell_1, \ldots, \ell_n)$$

where ℓ is a static representation of a runtime object.

In this section, we introduce three specification constructs based on relationships. The first construct, *relationship effects*, specify how framework operations change associations between objects. The second construct, *constraints*, uses relationships to specify the non-local constraints on framework operations. Finally, *relation inference rules* specify how relationships can be inferred based on the current state of other relationships, regardless of what operations are used.

2.1 Relationship Effects

Relationship effects specify changes to the relations that occur after calling a framework method. The framework developer annotates the framework methods with information about how the calling object, parameters, and return value are related (or not related) after a call to the method. These annotations describe additions and removals of relationships from a relation. For example, the annotation @Item({item, list}, ADD) creates an "Item" relationship between `item` and `list`, while @Item({item, list}, REMOVE) removes this relationship[2]. Relationship effects may refer to the

[2]We are presenting a simplified version of the syntax for readability purposes. The correct Java syntax for the add annotation appears as @Item(params={"item", "list"}, effect=ADD). This is the syntax used in the implementation.

is triggered, then this predicate must be true under the current relationship context. If the requires predicate is not true, this is a broken constraint and the analysis should signal an error in the plugin.

4. *effect list*: This is a list of relationship effects. These effects will only be applied if the constraint is triggered.

In the first example at the top of Listing 7, the constraint is checking that at every call to `ListItem.setSelected(boolean)`, if the relationship context shows that the argument is false, the receiver is a `Child` of a `ListControl`, and if the `ListControl` is a `DropDownList`, then it must also indicate that the `ListItem` is `Selected`. Additionally, the context will change so that the `DropDownList` is not `CorrectlySelected`. The second constraint is similar to the first and in enforces proper selection of `ListItem`s in a `DropDownList`. The third constraint ensures that the method does not end in an improper state by utilizing the "end-of-method" instruction to trigger when a plugin callback is about to end.

In some cases, the relationships between objects are implicit. Consider the `ListItemCollection` from the `DropDownList` example. In this example, the framework developer would like to state that items in this list are in a `Child` relation with the `ListControl` parent. However, it does not make sense to annotate the `ListItemCollection` class with this information since `ListItemCollection`s should not know about `ListControl`s.

2.3 Inferred relationships

In some cases, the relationships between objects are implicit. Consider the `ListItemCollection` from the `DropDownList` example. In this example, the framework developer would like to state that items in this list are in a `Child` relation with the `ListControl` parent. However, it does not make sense to annotate the `ListItemCollection` class with this information since `ListItemCollection`s should not know about `ListControl`s.

Inferred relationships describe these implicit relationships that can be assumed at any time. In Listing 7, lines 23-26 show an example for inferring a `Child` relationship based on the relations `ListItemCollection`s and `ListControl`s. Whenever the relationship context can show that the "trigger" predicate is true, it can infer the relationship effects in the "infer" list. It is possible to produce inferred relationships that directly conflict with the relationship context. To prevent this, the semantics of inferred relationships is that they are ignored in the case of a conflict, that is, relationships from declared relationship effects and constraints have a higher precedence.

3 The Relation Analysis

We have designed and implemented a static analysis to track relationships through plugin code and check plugin code against framework constraints. The relation analysis is a branch-sensitive, forward dataflow analysis[3]. It is designed to work on a three address code representation of Java-like source. We assume that the analysis runs in a framework that provides all of these features. In this section, we will present the analysis data structures, the intuition behind the three variations of the analysis, and a discussion of their tradeoffs. Section 4 defines how the analysis runs on each instruction.

The relation analysis is dependent on several other analyses, including a boolean constant propagation analysis and an alias analysis. The relation analysis uses the constant propagation analysis for the TEST effect. For this purpose, the relation analysis assumes there is a function $\mathcal{B}$ to which it can pass a variable and learn whether the represented value is true, false, or unknown.

The relation analysis can use any alias analysis which implements a simple interface. First, it assumes there is a context $\mathcal{L}$ that given any variable $\mathtt{x}$, provides a finite set $\bar{\ell}$ of abstract locations that the variable might point to. Second, it assumes a context Γ_ℓ which maps every location ℓ to a type τ. The combination of these two contexts, $< \Gamma_\ell, \mathcal{L} >$ is represented as the alias lattice $\mathcal{A}$.

The alias lattice must be conservative in its abstraction of the heap, as defined by Definition 1.

Definition 1 (Abstraction of Alias Lattice)**.** *Assume that a heap* $\mathtt{h}$ *is defined as a set of source variables* $\mathtt{x}$ *which point to a runtime location* ℓ *of type* τ*. Let* H *be all the possible heaps at a particular program counter. An alias lattice* $< \Gamma_\ell, \mathcal{L} >$ *abstracts* H *at a program counter if and only if*

$$
\begin{array}{l}
\forall\, \mathtt{h} \in \mathsf{H} \,.\, \mathtt{dom}(\mathtt{h}) = \mathtt{dom}(\mathcal{L}) \text{and} \\
\quad \forall\, (\mathtt{x}_1 \,,\!\!\rightarrow \ell_1 : \tau_1) \in \mathtt{h} \,.\, \forall\, (\mathtt{x}_2 \,,\!\!\rightarrow \ell_2 : \tau_2) \in \mathtt{h} \,. \\
\qquad \text{if } \mathtt{x}_1 \neq \mathtt{x}_2 \text{ and } \ell_1 = \ell_2 \text{ then} \\
\qquad\quad \ell' \in \mathcal{L}(\mathtt{x}_1) \text{ and } \ell' \in \mathcal{L}(\mathtt{x}_2) \text{ and } \tau_1 <: \Gamma_\ell(\ell') \\
\qquad \text{and} \\
\qquad \text{if } \mathtt{x}_1 \neq \mathtt{x}_2 \text{ and } \ell_1 \neq \ell_2 \text{ then} \\
\qquad\quad \ell'_1 \in \mathcal{L}(\mathtt{x}_1) \text{ and } \ell'_2 \in \mathcal{L}(\mathtt{x}_2) \text{ and } \ell'_1 \neq \ell'_2 \text{ and } \tau_1 <: \Gamma_\ell(\ell'_1) \text{ and } \tau_2 <: \Gamma_\ell(\ell'_2)
\end{array}
$$

This definition ensures that if two variables alias under any heap, then the alias lattice will reflect that by putting the same location ℓ' into each of their location lists. Likewise, if any heap can determine that the two variables are not aliased, then the alias lattice will reflect this possibility as well by having a distinct location in each location set. The definition also ensures that the typing context Γ_ℓ has the most general type for a location.

As long as the alias analysis maintains the abstraction property and can provide the required interface, the relation analysis can be proven to be either sound or complete. Of course, a more precise alias analysis will increase the precision of the relation analysis.

[3]By branch-sensitive, we mean that the true and false branches of a conditional may receive different lattice information depending upon the condition. The transfer function on the condition is called twice, once assuming that the result is false, and once assuming that it is true. This is not a path-sensitive analysis; the branch condition is not saved for use after the branches merge together.

3.1 The Relationship State Lattice

We track the status of a relationship using the four-point dataflow lattice represented in Figure 3, where `unknown` represents either true or false and `bottom` is a special case used only inside the flow function. The relation analysis uses a tuple lattice which maps all relationships we want to track to a relationship state lattice element. We will represent this tuple lattice as ρ. We will say that ρ is *consistent* with an alias lattice $\mathcal{A}$ when the domain of ρ is equal to the set of relationships that are possible under $\mathcal{A}$.

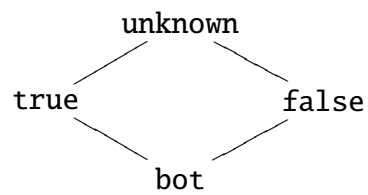

Figure 3: The simple lattice for a relationship

Notice that as more references enter the context, there are more possible relationships, and the height of ρ grows. Even so, the height is always finite as there is a finite number of locations and a finite number of relations. As the flow function is monotonic, the analysis always reaches a fix-point.

3.2 Flow Function

The analysis flow function is responsible for two tasks; it must check that a given operation is valid, and it must apply any specified relationship changes to the lattice. The flow function is defined as

$$f_{\mathcal{C};\mathcal{A};\mathcal{B}}(\rho, \mathtt{instr}) = \rho'$$

where $\mathcal{C}$ are all the constraints, $\mathcal{A}$ is the alias lattice, $\mathcal{B}$ is the boolean constant lattice, ρ is the starting relation lattice, ρ' is the ending relation lattice, and $\mathtt{instr}$ is the three-address code instruction on which we are running the analysis. The analysis goes through each constraint in $\mathcal{C}$ and checks for a match. It first checks to see whether the operation defined by the constraint matches the instruction, thus representing a syntactic match. It also checks to see whether ρ determines that the trigger of the constraint applies. If so, it has both a syntactic and semantic match, and it binds the specification variables to the locations that triggered the match.

Once the analysis has a match, two things must occur. First, it uses the bindings generated above to show that the required predicate of the constraint is true under ρ. If it is not true, then the analysis reports an error on $\mathtt{instr}$. Second, the analysis must use the same bindings to produce ρ' by applying the relationship effects.

3.3 Soundness and Completeness

Soundness and completeness allow the user of the analysis to either have confidence that there are no errors at runtime if the analysis finds none (if it is sound) or that any errors the analysis finds will actually occur in some runtime scenario (if it is complete). For the purposes of these definitions, an error is a dynamic interpretation of the constraint which causes the requires predicate to fail. In the formal semantics, an error is signaled as a failure for the flow function to produce a new lattice for a particular instruction.

We define soundness and completeness of the relation analysis by assuming an alias analysis which abstracts the heap using $\mathcal{A}$, as described above. For both of these theorems, we let $\mathcal{A}^{\mathtt{conc}}$

Table 1: Differences between sound, complete, and compromise variant

	Trigger Predicate checks when...	Requires Predicate passes when...
Sound	True or Unknown	True
Complete	True	True or Unknown
Compromise	True	True

define the actual heap at some point of an real execution, and we let $\mathcal{A}^{abs}$ be a sound approximation of $\mathcal{A}^{conc}$. We also let ρ_{abs} and ρ_{conc} be relationship lattices consistent with $\mathcal{A}^{abs}$ and $\mathcal{A}^{conc}$ where ρ_{abs} is an abstraction of the concrete runtime lattice ρ_{conc}, defined as $\rho_{conc} \sqsubseteq \rho_{abs}$.

If the relation analysis is sound, we expect that if the flow function runs to completion using the imprecise lattice ρ^{abs}, then any more concrete lattice will also run to completion for that instruction. As the flow function only runs to completion if it finds no errors, then there may be false positives from when ρ^{abs} produces errors, but there will be no false negatives. To be locally sound for this instruction, the analysis must also produce a new abstract lattice that conservatively approximates any new concrete lattice. Theorem 3.1 captures the intuition of local soundness formally. Global soundness follows from local soundness, the monotonicity of the flow function, and the initial conditions of the lattice.

Theorem 3.1 (Local Soundness of Relations Analysis)**.**
if $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}}(\rho^{abs}, \mathtt{instr}) = \rho^{abs\prime}$ and $\rho^{conc} \sqsubseteq \rho^{abs}$
then $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}}(\rho^{conc}, \mathtt{instr}) = \rho^{conc\prime}$ and $\rho^{conc\prime} \sqsubseteq \rho^{abs\prime}$

If the relation analysis is complete, we expect a theorem which is the opposite of the soundness theorem and is shown in Theorem 3.2. If a flow function runs to completion on a lattice ρ^{conc}, then it will also run to completion on any abstraction of that lattice. An analysis with this property may produce false negatives, as the analysis can find an error using the concrete lattice yet run to completion on the abstract lattice, but it will produce no false positives. Like the sound analysis, the results from the flow function must maintain their existing precision relationship.

Theorem 3.2 (Local Completeness of Relations Analysis)**.**
if $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}}(\rho^{conc}, \mathtt{instr}) = \rho^{conc\prime}$ and $\rho^{conc} \sqsubseteq \rho^{abs}$
then $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}}(\rho^{abs}, \mathtt{instr}) = \rho^{abs\prime}$ and $\rho^{conc\prime} \sqsubseteq \rho^{abs\prime}$

The relation analysis can be either sound, complete, or a compromise of the two, by making only minor changes to the analysis. Proofs of soundness and completeness, for the sound and complete variants respectively, can be found in the appendicies. The differences between the variants are summarized in Table 1 and are described below.

Trigger condition. The trigger predicate determines when the constraint will check the required predicate and when it will produce effects. The sound analysis will trigger a constraint whenever there is even a possibility of it triggering at runtime. Therefore, it triggers when the predicate is either true or unknown. The complete variant can produce no false positives, so it will only check the requires predicate when the trigger predicate is definitely true. Regardless of the variant, if the trigger is either true or unknown, the analysis produces a set of changes to make to the lattice based upon the effects list.

```
public class ListItemCollection {
    @Item( , thisg, REMOVE)
    public void clear() {...}
    ...
}
```

```
@Constraint(
    op = "ListItemCollection.clear()",
    trigger = "x instanceof ListItem",
    requires = "true",
    e ect = f "!Item(x, this)" g
)
```

Figure 4: Translating a relation effect with wildcards into a constraint

Error condition. The requires predicate should be true to signal that the operation is safe to use. The sound variant will cause an error whenever the required predicate is false or unknown. The complete variant, however, can only cause an error if it is sure there is one, so it only ags an error if the requires predicate is de nitely false.

Table 1 also shows a variant of the analysis that, while neither sound or complete, we believe is a good compromise between the two. The compromise variant attempts to minimize the number of false positives and false negatives by only triggering when the trigger predicate is de nitely true, but then signaling an error if the requires predicate is either false or unknown. While this version can produce false positives and false negatives, we believe it will be the most cost-effective compromise in practice, based on our experience described in Section 5. Additionally, this version may utilize inferred relations, a feature which is inherently neither sound or complete, but reduces the speci cation burden on the framework developer.

4 Abstract Semantics

In this section, we present formal semantics for a simpli ed version of the speci cations and analysis, the grammar for which is shown in Figure 5. We do not specialized relations for equality(==) and typing (instanceof). It is possible to add specialized relations by calling out to other ow analyses in the same manner as is done with both the boolean constant propagation analysis and the alias analysis.

Relation effects and wildcards are both syntactic sugar that can be easily translated into a constraint form. Relation effects are translated by considering them as a constraint on the annotated method with a true trigger predicate, a true requires predicate, and the effect list as annotated. Wildcards are easily rewritten by declaring a fresh variable in the trigger predicate and constraining it to have the desired type. Figure 4 shows an example effect with a wildcard translated into a constraint.

The lattice has the usual operators of join ( ) (and precision ( ), which work as expected for a tuple lattice. We also introduce three additional operators, de ned in Figure 6. Equivalence join ( ) will resolve to unknown if the two sides are not equal. Overriding meet ( ) has the property that if the right side has a de ned value (not), then it will use the right value, otherwise it will use the left value. The polarity operator ( ) will push all non-bottom values to the top of the lattice. Finally, we also de ne $?_A$ as a special lattice which is consistent with the alias lattice and which maps every relationship to .