Proper Plugin Protocols

Ciera N.C. Jaspan

October 3, 2011
CMU-ISR-10-???

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich (Chair)
Mary Shaw
William Scherlis
Gary Leavens (University of Central Florida)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Abstract

Software frameworks provide great reuse benefits to the software engineering community. Plugins can reuse large parts of the framework and provide a rich application with only a few lines of code. However, frameworks frequently require plugin developers to be aware of complex protocols between objects, and improper use of these protocols causes exceptions and unexpected behavior at run time. This dissertation introduces *collaboration constraints*, rules governing how multiple objects may interact in a complex protocol. These constraints are particularly difficult to understand and analyze because they may extend across type boundaries and even programming language boundaries. In this dissertation, I have done an empirical study of framework help forums which showed that collaboration constraints are burdensome for developers, as they take hours or even days to resolve. Additionally, I studied the designs of reusable components and found that collaboration constraints are essential to the designs of software frameworks.

This dissertation introduces Fusion, an adoptable specification language and static analysis for collaboration constraints. The specification language uses *relationships* to describe the abstract associations between objects and allows developers to specify collaboration constraints as logical predicates of relationships. Since relationships are an abstraction above code, this allows developers to easily specify constraints that cross type and language boundaries. There are three variants of the analysis: a sound variant that has false positives but no false negatives, a complete variant that has false negatives but no false positives, and a pragmatic variant that attempts to balance this tradeoff. In this dissertation, I successfully used Fusion to specify and analyze constraints from examples found in the help forums of the ASP.NET and Spring frameworks. Additionally, I ran Fusion on DaCapo, a 1.5 MLOC DaCapo benchmark for program analysis, to show that Fusion has the necessary properties to be adoptable by industry.

This dissertation examines many tradeoffs: the tradeoffs of framework designs, the tradeoffs of specification precision, and the tradeoffs of program analysis results are all featured. A central theme of this work is that there is no right solution to collaboration constraints; there are only solutions that work better for a particular instance of the problem.

dedicate stuff

Acknowledgments

Numerous people have...

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Object Protocols

In object-oriented programming, developers frequently use *protocols* to describe how the state of an object changes as operations are called on it. The canonical example is a `File` object, which transitions between states as seen in the state machine in Figure 1.1. In this protocol, the `read` operation cannot be called unless the object has been opened, and once opened, it must be closed for open to be called again. Another canonical example is an `Iterator`, seen in Figure 1.2. The client of the `Iterator` must always check the return value of `Iterator.hasNext()` before calling `Iterator.next()`. Object protocols such as the `File` and `Iterator` protocols have been well studied; a large body of research has been dedicated to discovering them using program analysis [67, 69, 85], specifying and checking them statically [16, 32, 65, 79] and dynamically [19, 20, 78, 112], and even raising them to the level of a programming abstraction [105].

While prior work has made tremendous strides, there has been a glaring problem: as said by Beck and Cunningham, "No object is an island." [13] Objects interact with other objects, and these multi-object interactions are governed by protocols more complex than ones for a single object. The canonical example here is that of the protocol between a `Collection` and its `Iterator`, as seen in Figure 1.3. In this protocol, an `Iterator` cannot be used after a modifying operation is called on the `Collection` (though read-only operations are fine). Prior research on specifying and statically checking protocols either cannot handle multiple objects or can only do so in a limited way [16, 20, 65, 78, 79].
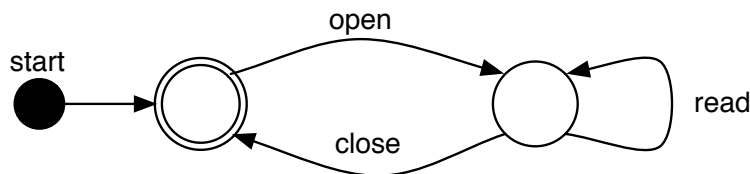
Figure 1.1: State machine of a typical `File` object protocol. The closed circle represents the start of the protocol. The open circles are states in the protocol, and the arrows represent the valid transitions from one state to the next. The doubled circle represents a valid end state for the protocol.
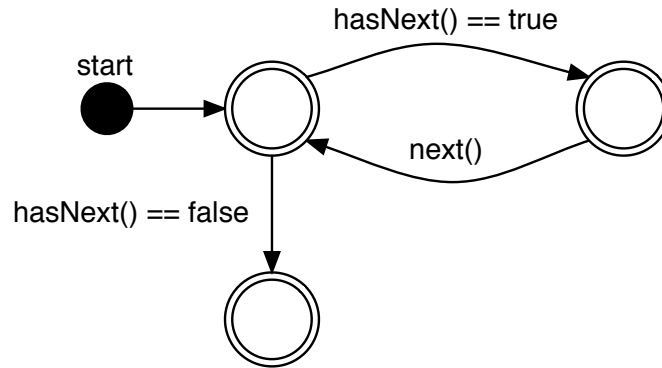
Figure 1.2: State machine of a typical `Iterator` object protocol. Notice that all the states are valid end states.
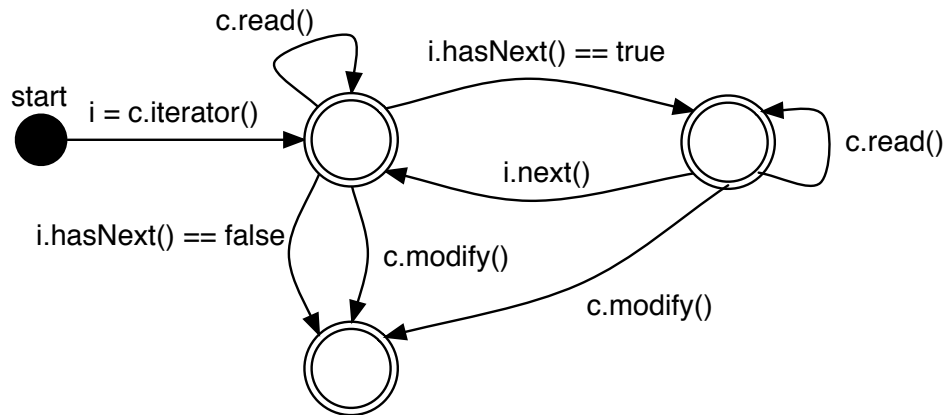
Figure 1.3: State machine of a typical protocol with a `Collection` and an `Iterator`.

Research cannot afford to ignore these protocols. While they might not appear frequently in small, stand-alone programs, they are common in reusable components such as software frameworks. The designs of these components seek to maximize the reuse of the component, both in terms of amount of functionality and in terms of the number of possible clients and the flexibility to address the diverse needs of those clients. As we will see in later chapters, multi-object constraints will occur more frequently in these situations. Additionally, these multi-object constraints are significantly more difficult to understand and fix. While we might expect a developer to follow the state machines in Figures 1.1 and 1.2, or even Figure 1.3, the state machine that results from more objects get very complex; Figure 1.4 provides one such example.

1.1   This Dissertation

In this dissertation, I will refine the concept of a multi-object protocol as a *collaboration constraint*. A collaboration constraint is a state-based restriction on how multiple objects may interact. Multi-
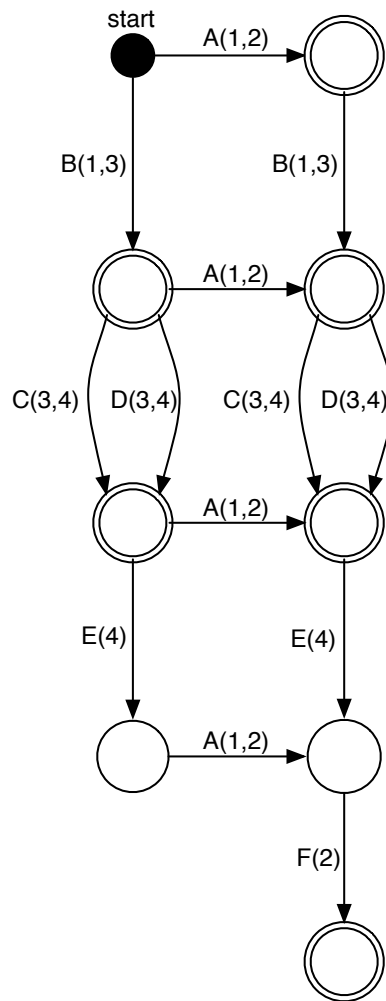
Figure 1.4: An abstraction of a complex multi-object protocol, from the example presented later in Vignette 3.1 of the ASP.NET framework. This protocol has six relevant operations (A-F) across four objects (1-4). The operators and the objects they are parameterized on is shown in the transition label, thus A(1,2) is the A operator with objects 1 and 2 as parameters.

object protocols can be thought of as a series of collaboration constraints, though as we will see later, collaboration constraints occur at times which would not traditionally be called a protocol.

To help developers specify and analyze collaboration constraints, I have created a new abstraction called a *relationship*, which represents an abstract, named association between several objects. Using the concepts of collaboration constraints and relationships, this dissertation has the following thesis:

> *Collaboration constraints are inherent to the design of software frameworks but are burdensome for plugin developers. These constraints can be defined by specifications that describe the relationships between objects and how relationships change, and an adoptable static analysis can check that code conforms to the specified constraints.*

More specifically, the goals of this thesis are:

1. to determine whether there is an inherent or accidental reason for why software frameworks have collaboration constraints (Chapter 2),

2. to discover the properties that are common to collaboration constraints and must be addressed by any solution (Chapter 3),

3. to create a specification language that will allow framework developers to conscisely describe collaboration constraints (Chapters 4 and 5), and

4. to create a static program analysis that will allow a users of a framework to discover any broken collaboration constraints in their code and do so faster than current practice (Chapters 4 and 5).

As can be seen from the above goals, this work is a study of both a problem and a solution. The first two chapters will be dedicated solely to understanding the problem of software frameworks and collaboration constraints. These chapters will both use archival analysis and taxonomies to thoroughly understand the problem. To solve this problem, I have created the Fusion (Framework Usage SpecificatIONs) language and static analysis, which is described in detail in Chapters 4 and 5. This solution is meant to be adoptable by industry, and so I present two case studies to show that Fusion can specify and detect violations of the kinds of collaboration constraints found in industry (Chapter 6) and that there is evidence that this form of solution will be adoptable in practice, not just by researchers (Chapter 7). Finally, there has been a significant amount of related work in the areas of framework specifications, protocol specifications, and logical frameworks; this work will be covered in Chapter 8.

1.2   Research Contributions and Expectations

This research will make three primary contributions to research and industry:

1. *Collaboration Constraints and Frameworks.* Provide a precise and useful definition of a software framework and show that the collaboration constraints described are essential to the nature of frameworks.

(a) Present a clear and useful definition of frameworks and plugins, driven by industry constructs and designs rather than historical accident. The definition will not be limited to a particular design paradigm but will abstract over them in a useful manner.

(b) Present a clear analysis of the interactions between frameworks and plugins that can be used to drive further research in the field. In particular, it will provide a taxonomy of the types of constraints that frameworks impose on plugins based on empirical evidence. This taxonomy can be used to improve error prevention techniques, including better framework design and other analyses.

(c) Show that the collaboration constraints described are common in practice and are particularly problematic for plugin developers.

2. *Relationships and Fusion.* Show that relationships can be used to specify collaboration constraints that occur in Java and XML frameworks.

(a) Define the relationship abstraction and demonstrate its ability to specify collaboration constraints.

(b) Present a specification language to describe collaboration constraints that occur in frameworks based on Java and XML. While there are many languages to specify either Java or XML, there are no known specification languages that can describe design intent which spans across both languages.

3. *Fusion Analysis.* Present an adoptable static analysis of the specifications that can detect violated collaboration constraints in plugin code.

(a) Provide a static analysis which checks plugins for conformance to collaboration constraint specifications and directs the developers to the cause of any errors found.

(b) Present detailed case studies on the differences between three variants of the analysis: a sound version, a complete version, and a pragmatic version which is neither sound nor complete, but instead balances the tradeoffs of false positives and false negatives. The case studies detail several sources of imprecision for the static analysis, the affect of this imprecision on the three variants, and the extent to which this imprecision occurs in industry code.

I expect that a tool like Fusion would be primarily used by industry professionals to specify their frameworks and assist plugin developers with finding problems. In particular, I anticipate that framework developers would adopt this tool incrementally by adding relationship specifications on an on-demand basis; when a plugin developer asks about a constraint on the forum or mailing list, the framework developers can answer the question and then add specifications for that constraint in the next release. After the next release, plugin developers would be able to run the analysis to detect violations of these constraints without any assistance from other developers.

Many large frameworks, such as Spring and ASP.NET, have generated third-party service companies which sell developer tools and consulting services. I expect these companies would be attracted to this work as a means of increasing business; these service companies could sell specification sets and tools. As the number of constraints in a particular framework increase, I would

also expect framework vendors and service companies to build more tools that take advantage of these specifications. For example, a tool that visually describes the constraints would be a useful form of documentation, as would a tool that suggests operations based on the constraints which need to be satisfied.

Chapter 2

Software Frameworks

Software frameworks are extremely popular form of reuse in a variety of domains from web applications (ASP.NET [73], Spring [111], Ruby on Rails [31]) to parallel computing (Hadoop [9], OpenMPI [109]) to developer tools (Eclipse [106], JUnit [107]) to social networks (Facebook [35]). The popularity of software frameworks stems from the large reuse benefits that they provide to industry. With relatively few lines of code, software frameworks allow developers to create large and complex applications that are customized for a specific purpose, unknown to the developers of the framework.

While the reuse benefits that frameworks provide make them worthwhile despite high costs, they are notoriously difficult to use, design, and document. As described in Chapter 8, a plethora of work has been done previously to ease this burden. Unfortunately, few ideas have caught on, and software frameworks remain an unavoidable yet beneficial obstacle for software engineering.

In this chapter, I will not be presenting a way to make software frameworks easier to use; that topic will be reserved for later chapters. Instead, I will provide a better understanding for how it can be that software frameworks provide both high level of reuse yet form obstacles to achieving it, and I will also describe how this tradeoff works differently for other forms of software reuse, like libraries and toolkits. Most importantly, I will show that some amount of *un*usability is actually essential to the design of software frameworks in order to achieve the benefits we seek.

This investigation will start with an architectural understanding of software frameworks. From this, I will identify several quality attributes that are essential to framework designs and make software frameworks distinct from other forms of component-based reuse, such as libraries or tookits. We will see that the tradeoffs between these quality attributes lead to unusable APIs in software frameworks. Finally, the chapter will show how the relatively new practice of depending on declarative artifacts in framework designs has both provided further levels of reuse yet increased the complexity of these designs further than ever before.

Throughout this chapter, I will introduce and reference vignettes where a plugin developer is attempting to reuse a software framework. The vignettes in this chapter are from the ASP.NET web application framework, a software framework for creating a group of web pages that link together to form an application. These vignettes will illustrate several arguments within this chapter and will be referenced in later chapters.

2.1   An architectural definition of software frameworks

Software frameworks are known to be difficult to design and use, but what exactly makes some piece of software a framework? What makes a software framework different from other reusable components, like libraries and toolkits? How do they compare to product lines? In this section, I'll give an overview of several definitions of software frameworks, but I will ultimately argue for an architectural definition of software frameworks.

Software frameworks originally came from the object-oriented community, and as such, they were defined in OO terms.

> *A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact. [60]*

However, OO-based definitions are too narrow in practice; the term "framework" is now applied to software which uses non-OO mechanisms as the primary way to interact with the client code.[1]

Others have taken the approach that a software framework has an inherent property: *inversion of control*. Inversion of control means that the framework controls the flow of data and the flow of execution through the program. This is in contrast to a library, where the application calls the library and is in control of the execution and data. This idea that the framework "calls back" to the application is also known as the Hollywood Principle ("Don't call us; we'll call you") and is commonly found in descriptions of frameworks.

> *The Hollywood Principle is a key to understanding frameworks. It lets a framework capture architectural and implementation artifacts that don't vary, deferring the variant parts to application-specific subclasses. [110]*

However, this description is still not ideal, as callbacks are a common paradigm throughout software. For example, many collection libraries will sort a collection by calling back to a provided sort function, yet clearly this software does not have the complexity of those that we term software frameworks, like ASP.NET or Eclipse. Additionally, frameworks may not use callbacks for all features; frameworks are increasingly turning to in-code annotations and configuration files. Therefore, definitions based on inversion of control end up both excluding more modern frameworks, yet including simpler forms of reuse.

My view of software frameworks stems from software architecture concepts. In particular, software frameworks are not simply a module of code reuse, but a module of code that implements and enforces a software architecture. This view is shared by industry developers; the only definition I found which described frameworks in architectural terms was in the book "Software Factories", by two Microsoft employees [47]. [2] Likewise, I will define a framework, and the associated terms, in architectural vocabulary.

Definition 2 (Software Framework, or just Framework)**.** A *software framework* is a set of reusable modules which requires that their clients conform to a predefined architecture.

---

[1]Many framework designs retain some OO elements and use objects, however, inheritance is no longer the primary reuse mechanism.

[2]In this book, they say that "A framework is developed to bootstrap implementations of products based on a common architectural style." However, this definition is not quite right as a framework is not solely about bootstrapping.

Definition 3 (Plugin)**.** A *plugin* is a module which extends a framework and works within the constraints of a framework's defined architecture to add specific functionality. [3]

A framework is not simply a set of modules with a protocol for how to access some reusable functionality. In fact, a framework may have very little functionality; it may only be an implementation to connect plugins together. Regardless, the framework encapsulates the architecture for the final system. Consider the following examples:

- Open|SpeedShop [108] is a framework for creating distributed dynamic analyses. It has several types of plugins: wizards set up an experiment to run, collectors gather the data, aggregators put data together, analyses run some computation on the data, and views display the results to the user. While the framework does provide some functionality, its primary purpose is connecting these plugins into a pipe-and-filter architecture. In fact, the reusable functionality it provides is handled by some built-in libraries; the framework itself just loads components and connects them together.

- Eclipse [106] is a framework for developer tools. Eclipse provides a mechanism for plugins to define their own extension points, so that plugins in Eclipse can also be small frameworks and have their own plugins. Eclipse loads the plugins and connects them together in an architecture that resembles an acyclic graph of frameworks and plugins.

- Spring [111] is a framework for web applications. Each web application that uses Spring must adhere to a model-view-controller architecture. Like Open|SpeedShop, Spring provides some reusable functionality as well, but this functionality is packaged into libraries. In Spring, these libraries may also be plugins and can be replaced by other plugins.

- ASP.NET [73] is another framework for web applications, which also uses a model-view-controller architecture. Unlike Spring, ASP.NET requires complete buy-in to their framework with few alternative options for the given libraries. However, it also provides plugins with many points for variation within the given modules, as can be seen in Vignette 2.1.

Of course, frameworks are not the only form of reusable code. Other reusable codebases go by the names of *library* or *toolkit*.[4] While there is no fully agreed on definition for these terms either, they are frequently used to describe code which contains functional reuse, but not architectural reuse. For example, a collections library, an XML parsing library, and a UI controls toolkit all provide significant reusable functionality. However, using libraries and toolkits do not typically impact the architecture of the application; such libraries are used by applications from many domains and with very diverse architectures. While switching to a different collections library might require significant low-level code changes, it would not affect the architecture of the application.

---

[3] It is interesting to notice that a plugin may be developed by the person who is composing the plugin with the framework, by a third-party, or even by the framework developer. Who develops the plugin is a separate issue from what it is.

[4] In practice, these terms seem to be nearly interchangeable, though library generally implies a single cohesive component and toolkit implies a related set of smaller component.

While the four frameworks above have reusable functionality, large portions of the functionality could be replaced, or even removed, and what would remain would still be a software framework. In fact, any replaced functionality would still have to conform to the framework's architecture. All of these frameworks also use OO designs, but the designs are not purely object-oriented. The frameworks above heavily use configuration files, aspects, and dependency injection; objects are only a part of how they interact with plugins. Therefore I argue that a framework is not simply a set of modules with reusable, object-oriented functionality, or even a reusable object-oriented design. While a framework may contain OO designs, a framework is primarily a set of modules which encapsulates a reusable architecture.

Since a plugin must adhere to the architecture provided by the framework, *architectural mismatch*, as originally defined by Garlan, Allen, and Ockerbloom [46], is a serious problem for plugins. Vignette 2.1 provides an example where a plugin does not adhere to the given architecture and runs into problems. Plugin developers must take care to fully understand the architectural implications of using a particular software framework and the potential consequences of combining several frameworks in a single application. When viewed from an architectural perspective, it is no surprise that frameworks can be difficult to use, even for experienced developers, as they are a working example of one of the most difficult problems facing software engineering.

Plugin Vignette 2.1: Lifecycle

The ASP.NET web application framework allows developers to create a plugin that corresponds to a web page in a web application. By creating several plugins and connecting them together with links, they create a complete web application. When a user requests a web page, an HTTP request is sent to access that page, and the framework uses the provided plugin to generate the HTML for the page and return it back to the user.

At the highest level of abstraction, the ASP.NET framework uses a stateless client-server architecture to interact with the user. This architecture is abstracted as much as possible from the plugins, to the level that plugins can even pretend to be stateful because the server handles the storing and reloading of state. Any use of this stateful abstraction must be done through the framework provided mechanisms and according to a given protocol. Otherwise, the plugin is not aware of, and has no control over, the client-server architecture.

There is a lower-level architecture that the plugin must be aware of: the ASP.NET framework requires plugins to adhere to a model-view-controller architecture. All plugins must conform to this architecture and are composed of three pieces:

- *View* The plugin provides an ASPX file which represents a static view of the web page. ASPX is HTML with features specific to ASP.NET, and the framework will process this file into raw HTML later.

- *Model* The framework creates the model based upon the HTTP request from the user for a page and saved state from prior requests. The plugin can change this model in the controller.

- *Controller* The plugin provides a "code-behind" class, written in either C# or VB.NET, that defines events that happen in response to user actions. Additionally, this controller can dynamically change the view and the model through a series of callbacks from the server, as described in more detail below.

To create the HTML for a user request, the ASP.NET framework processes the ASPX file into HTML. This is a multi-step process, and while this process takes place, the framework makes a series of calls to the

code-behind class. This series of calls is known as the page lifecycle, and it occurs on every user request of a page. Lifecycle calls allow the plugin to perform dynamic modifications to the page. For example, the code-behind class can use the callbacks to populate values to the controls or even dynamically add or remove controls.

The most commonly used lifecycle methods are `PreInit`, `Init`, and `Load` (though there are eight others that can be used). PreInit is called before the framework begins processing the ASPX, so the controls on the page are not initialized yet. Init is called after the controls are initialized from the ASPX, but before they are loaded with their stateful data. Load is called after the framework has loaded stateful data back into the controls.

It's very important for developers to understand how this lifecycle works, as misusing the lifecycle results in null references [91], disappearing controls [102], and missing user input [11]. Each of these problems was seen on the ASP.NET help forums, and like many others, the posters of the problems were instructed to read the Page Lifecycle documentation [74].

As an example of how misusing the lifecycle results in unusual problems, consider the code in Listing 2.1 from the ASP.NET help forums. The purpose of this code is to set the initial values in the drop down list called `DateYear`, which is defined in the associated ASPX file. However, the code was throwing an null reference exception at line 15.

Listing 2.1: Incorrect usage of the page lifecycle

```vb
1   Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.PreInit
2
3       'Generate years for drop down menu
4       Dim Dates As New Collections.Generic.List(Of System.DateTime)
5       'Dates.Add(System.DateTime.Now)
6
7       If Not Me.IsPostBack Then
8           ' Add next 5 years
9           For i As Integer = 0 To 4
10              Dates.Add(System.DateTime.Now.AddYears(i))
11          Next
12      End If
13
14      ' DateYear is a statically declared DropDownList
15      Me.DateYear.DataSource = Dates
16      Me.DateYear.DataTextField = "Year"
17
18      Me.DateYear.DataBind()
19  End Sub
```

Three other developers responded with possible problems in the code, but each potential issue they raised turned out to be implemented correctly. Finally, the third developer found the mistake on line 1 of Listing 2.1.

Sorry just noticed the event you are using! PreInit. You should be using init for this.

You need to read the page life cycle overview `http://msdn2.microsoft.com/en-us/library/ms178472.aspx`

CreateChildControls will be called on the control between these two events.

As described earlier, the PreInit callback happens before any controls are initialized, so the field `DateYear` is still null. However, the Init callback will guarantee that all statically declared controls exist,

though they have no data yet, and is the appropriate place to load this data. In several other forum postings, developers confused the Init and Load events, which results in either no data (if the developer created controls in Load, after the data loading occurred) or null references and clobbered data (if the user attempt to read or write the control's data while in the Init callback, before data loading occurred).

Each of these problems occurred not because of a simple coding error, but because the plugin developer misunderstood the architectural implications of using the framework. The plugin developer had to be aware not just of the available method calls and the local pre- and post-conditions, but also how these methods are used in the more global architecture. The plugin developer must be aware that in ASP.NET, they are buying into a stateless client-server architecture that will represent statefulness through a model-view controller sub-architecture. Not adhering to these architectural considerations and tradeoffs results in defective plugins.

2.2   The essential complexity of software frameworks

With an architectural definition of software frameworks in hand, the questions of why software frameworks are difficult to design, document, and use becomes more tractable. Software frameworks are difficult to design, document, and use because of the essential complexity of building code that encapsulates a reusable software architecture.

Since the goal of a software framework is to create a reusable software architecture, it must be able to embed many quality attributes tradeoffs into its design. This of course holds true for any software architecture design: the designer must carefully weight the tradeoffs between several quality attributes according to the purpose and goals of the system [12].

Designing a good architecture is know to be difficult, but the problem is compounded in the case of software frameworks. In addition to considering the quality attributes demanded by the domain of the software framework, all reusable modules have three additional quality attributes to consider. These three quality attributes can be thought of as three aspects of *reusability*. In addition to being defined below, the extreme ends of these quality attributes are depicted graphically in Figure 2.1.

- *Usability* is the ease of reusing the API of the module. For a module to have high usability, it must have a simple, well defined API with as few points of variation as possible [62]. While usability is relative to an individual's experience, one module might still be considered more usable than another, by both novices and experts alike.

- *Utility* is the amount of reuse achieved by a single reuser of the module. For a module to have high utility, it must provide as much reusable code as possible for applications that use it. This includes code for functional reuse and code for architectural reuse.

- *Versatility* is the scope of potential reusers of the module. For a module to have high versatility, it must be reusable by as many potential applications as possible, including future, unanticipated applications. To do this, it must be highly flexible so that it can be modified and reused by a wide range of applications.

(a) High Usability: The API is simple and easy to understand.

(b) Low Usability: The API is complex to use.

(c) High Utility: The client gets a large amount of reuse.

(d) Low Utility: The client receives relatively little reuse.

(e) High Versatility: There are many potential clients.

(f) Low Versatility: There are a few predetermined clients.
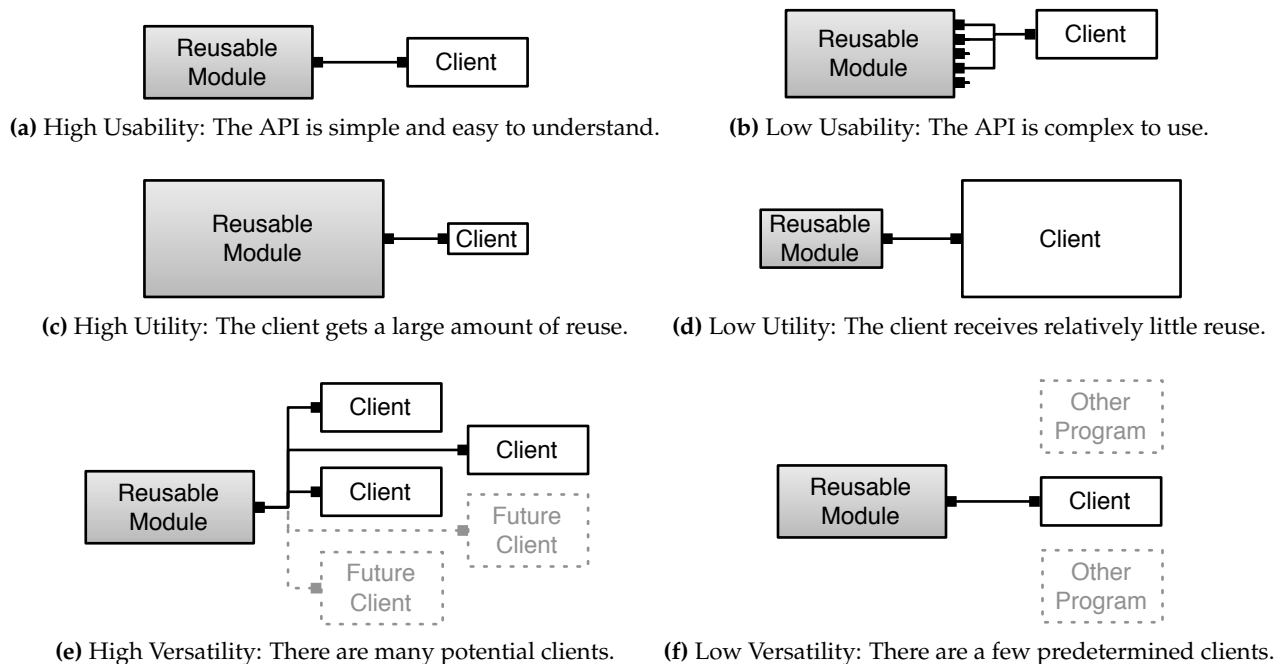
Figure 2.1: Graphic depiction of the extremes of usability, utility, and versatility for a reusable component. The left column depicts high levels of the quality attributes, while the right column depicts low levels.

Clearly, it is desirable for a reusable module to have high levels of usability, utility, and versatility in order to maximize its impact on the world (and consequently, its profit margins). However, even without considering other desirable quality attributes from the domain, these three are in conflict with one another. Figure 2.2 illustrates the tradeoff space, with examples of reusable modules that select different tradeoffs. While it is not possible to maximize all three of these quality attributes, Figure 2.2 shows the ways that two of these quality attributes can be met in a reusable module.

Region 1 represents libraries and toolkits, such as the Java Collection and I/O libraries. Such libraries are intended to be easy to use and to be reused by as many applications as possible (high usability and high versatility). However, they each provide a limited scope of features, such that a developer must add a lot to make a complete application (low utility).

Region 2 represents product line systems, such as those that appear internally within a company to be reused in all their systems. Like frameworks, product lines impact the architecture of the clients for the purpose of increasing utility. These systems are designed to be easy to use so that training costs are low, and they are designed to provide significant amounts of reuse for those products within the scope of the company's interests (high usability and high utility). However, as the product line would never be used outside the company, they can tightly control the scope of applications which might reuse the product line (low versatility).

Region 3 represents software frameworks such as those described earlier in this chapter and
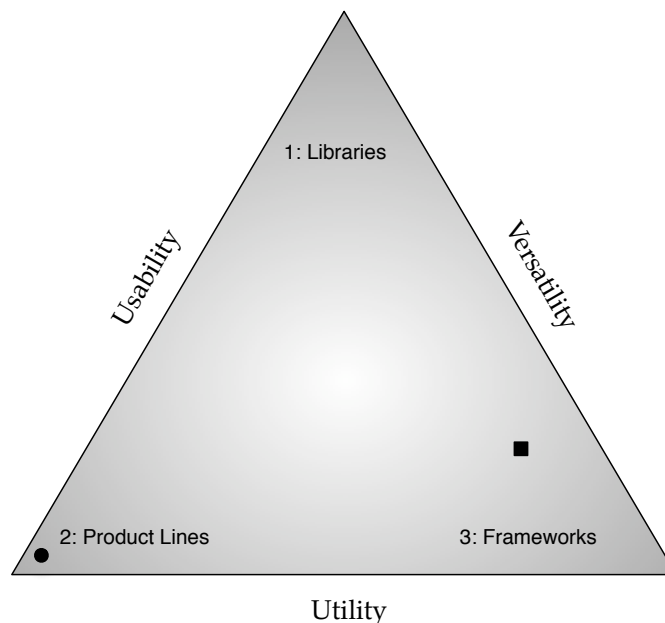
Figure 2.2: The tradeoff space for the quality attributes of usability, utility, and versatility for a resusable module. The small circle represents Ruby On Rails using their built-in scripts to create web applications, the small square represents using Ruby on Rails without the scripts.

throughout this thesis. In order to increase their impact in software, many of them aim to be as general purpose as possible (high versatility) and to provide extraordinarily high levels of utility. While the cost of this is low usability, this is deemed worthwhile if the users are expected and willing to stick through the steep learning curve and become a member of a community that continues to use the framework for years.

Any attempt to maximize all three quality attributes is bound to fail. Consider the case of a module with a small, highly usable API. If this module has maximized utility as well, then there is a lot of code behind that API. Of course, this code cannot be customized in any way, as allowing that would necessarily make the API more complex, so the module can only be used by a few clients that wish to reuse it as-is.

Let us try again from another approach: we can imagine a module, again with a small API, that is highly versatile and can be reused by many clients. To do this though, it must not be able to provide much functionality, as each added feature would increase the size of the API in order to give all clients the ability to customize it. The only way for a module to be usable and versatile is to provide relatively little utility.

It is important to note that the tradeoff with usability exists regardless of a programmers experience or of the particular abstractions available in today's programming languages. While an experienced developer might find the Collections library more usable than a student would, both expert and novice will find Eclipse to be a relatively less usable framework. Likewise, new abstractions in programming languages may increase the usability of all applications. However, as Eclipse attempts to maximize both utility and versatility, it will always be less usable than the Collections library, regardless of the abstraction chosen. A new abstraction (like object-oriented

programming, architectural styles, and many others) may shift the entire design space to make it all easier to use, but the core tradeoff, though weakened, will remain.

The tradeoff space in Figure 2.2 is not a discrete space and is somewhat blurry. For example, a reusable component may have sub-components that exist in different parts of the space when viewed by themselves; many frameworks and product lines contain internal libraries. Additionally, a component may shift location in this space depending on how it is reused. As an example, consider Ruby on Rails, a web application framework. The developers of this framework brag about being able to create a web application in only 15 minutes [50]. Unfortunately, the scope of possible web applications that can be made in this way is limited to a predefined set, so Ruby on Rails exists in the lower left corner alongside product-line systems. However, Ruby on Rails does provide a different mechanism for creating more complex web applications; doing this requires both more code and uses more complex APIs, so the tradeoffs of Ruby on Rails shift to the right. In cases like this, it's possible to think of the component as actually having two separate APIs; one for beginner use and one for expert use. This is fairly common for reusable components and can be seen in both the Swing GUI framework [104] and the Crystal static analysis framework [83].

The result this tradeoff is that frameworks are inherently difficult to use, even when designed well. If the designer of a framework made the decision to create a reusable software architecture that can be reused by a wide variety of applications and provide them with maximum reuse benefit, it is no wonder when the framework suffers from usability problems. Given these tradeoffs, software frameworks will always exist because of their utility and versatility for reuse, but we will have to live with the usability consequences. In this thesis, I will be addressing this issue by providing a program verification technique that will help plugin developers find the defects that occur as a result of a difficult to use API.

2.3 An added twist: declarative artifacts

There is one additional twist to current software frameworks that will be relevant for this thesis. Traditionally, software frameworks have used object-oriented programming techniques as the primary abstraction for reuse and communication with plugins. In recent years, declarative artifacts have become a popular secondary abstraction: Eclipse, ASP.NET, and Apache Server all require their plugins to create declarative artifacts. At first glance, these declarative artifacts do not even appear to be program code, and they might be considered a non-code artifact similar to image resources or translations for internationalization. In fact, as these declarative files might contain data specific to a particular runtime environment, in some circumstances they might not even be checked into a code repository.

How prevalent are these declarative artifacts? In a study done with Kevin Bierhoff, George Fairbanks, and Jonathan Aldrich, we found 11 industry frameworks that are using declarative artifacts; the full list of 17 frameworks that we studied can be found in Table 2.1. Declarative artifacts were used for a wide variety of purposes, including user interfaces, architecture configuration at runtime, descriptions of data formats and validation, deployment configuration, and server configuration. In all of these cases, a pure OO design would have not met the needs of the system, though some of the frameworks still provide the OO mechanisms.

Table 2.1: Summary of results from an archival analysis of 17 industry frameworks.

Framework	Language	Declarative Artifacts?
Apache HTTP Server	C	Yes
Applets	Java	No
ASP.NET	C#, VB.NET	Yes
AWT/Swing	Java	No
Corba	Various	No
Eclipse	Java	Yes
Enterprise Java Beans	Java	Optional
Facebook	PHP and Various	Yes
JUnit (and related)	Various	No
MFC	C++	Yes
OpenMPI	C	No
Ruby on Rails	Ruby	Yes
Servelets	Java	Yes
Spring	Java	Yes
WebOjbects	Java	Yes
WinForms	C# and Various	Yes
XSever	C	No

Declarative artifacts allow for additional modifiability that is not offered by traditional programming abstractions. In particular, they allow modifiability through time, through environments, and through the modifier. I will explain each of these concepts in turn.

Modifications through time.   As declarative artifacts are not evaluated until run time, they can be modified without recompiling. This allows for certain, predefined modifications (like the location of a database) to be easily made post-compilation.

Modifications of the environment.   Because declarative artifacts are modifiable through time and are not tied to program code, they can be modified separately for each environment that the system is deployed in. Following the database example once again, we can quickly deploy a system to multiple locations, without modifying any code, by editing a declarative artifact that specifies the location of the database for the particular deployment environment. This enables a company to develop and deploy complex product line systems with relatively little effort.

Modifications from unusual modifiers.   Finally, declarative artifacts can be created for specific non-programmers so they can modify the program without accessing the program code. These non-programmers might include UI designers, IT administrators, or even end-users. Using declarative files, each of these people can complete their modification task with little involvement of a software developer. In the database example, the declarative artifact can be changed by an IT administrator in response to new changes in the deployed environment. As a further example,

Vignette 2.2 features an ASP.NET plugin that uses a declarative artifact for the user-interface. In my experience at LEVEL Studios, this allowed the UI designers to modify the design of the web page in parallel with the software developer creating the functionality.

As practical as these declarative artifacts are for modifiability, they are not addressed in any known general purpose program verification systems.[5] As seen in Vignette 2.2, these artifacts are tied with the code to the extent that verifying the code alone is not useful. As part of this thesis, I will also be providing a first attempt at verifying declarative artifacts alongside program code.

Plugin Vignette 2.2: LoginView

On the ASP.NET forums, a developer reported that he was attempting to retrieve a `DropDownList` within his code-behind file, but his code was throwing a `NullReferenceException` [94]. His plugin uses a `LoginView` control, which allows developers to display some controls if the user is logged in, and other controls if the user is not logged in. It achieves this by having two *templates* which represent these states, as shown in the developer's ASPX file in Listing 2.2.

Listing 2.2: ASPX with a `LoginView`

```
1   <asp:LoginView ID="LoginScreen" runat="server">
2     <AnonymousTemplate>
3        You can only set up your account
4        when you are logged in.
5     </AnonymousTemplate>
6     <LoggedInTemplate>
7        <h4>Location</h4>
8        <asp:DropDownList ID="LocationList"
9            runat="server"/>
10       <asp:Button ID="ContinueButton"
11          runat="server" Text="Continue"/>
12     </LoggedInTemplate>
13   </asp:LoginView>
```

Listing 2.3: Incorrect way of retrieving controls in a `LoginView`

```
1   LoginView LoginScreen;
2
3   private void Page_Load(object sender, EventArgs e)
4   {
5      if (!isPostback()) {
6         DropDownList list = (DropDownList) LoginScreen.FindControl("LocationList");
7         list.DataSource = ...;
8         list.DataBind();
9      }
10  }
```

The developer properly set up a `LoginView`, including the `DropDownList` within it, in the ASPX file. The developer then went to his code-behind file in Listing 2.3, and in the initialization event, attempted to set up the `DropDownList` with data when the page is viewed for the first time. The typical way to get a sub-control

---

[5][7] addresses them for Ruby on Rails, but this solution is specific to the Ruby on Rails framework. Likewise, [100] provides simple verification for Spring.

is to call `Control.findControl` with the appropriate name; `findControl` will return null only if there is no sub-control with that name. While line 7 was throwing a `NullReferenceException` since `list` was null, the developer was confused because he had used exactly the name he declared in the ASPX file.

Another developer responded to the post and explained this unusual error. The original developer did correctly set up his controls so that the `DropDownList` would only show when the user is logged in. However, the `LoggedInTemplate` does more than just make the controls invisible if no user is logged in; the controls *will not even exist in memory unless a user is logged in*. Therefore, if a developer wishes to set up data in these controls, he must do so before the control is displayed, but only if the user has logged in. This constraint make more sense from a security perspective; we do not want any chance of the data within that control leaking out of the system, so it does not exist at all until necessary. The solution proposed was to first check the login status from `Request.isAuthenticated()`, using the page's `Request` object, as shown in the corrected Listing 2.4.

Listing 2.4: Correct way of retrieving controls in a `LoginView`

```
1   LoginView LoginScreen;
2
3   private void Page_Load(object sender, EventArgs e)
4   {
5      Request myRequest = getRequest();
6      if (!isPostback() && myRequest.isAuthenticated()) {
7         DropDownList list = (DropDownList) LoginScreen.FindControl("LocationList");
8         list.DataSource = ...;
9         list.DataBind();
10     }
11  }
```

This example quickly becomes more complex if we want to show different controls to different kinds of users. The `LoginView` also allows us to do this by creating many `RoleGroups` and associating each with user role, as shown in Listing 2.5. If we also want this functionality, we must check the properties of the logged-in user (Listing 2.6) to determine whether a control is accessible. This adds a great deal of complexity to the plugin, and it is compounded if a user is specified in more than one `LoginTemplate`.

Listing 2.5: ASPX with a `LoginView` and multiple `RoleGroups`

```
1      <asp:LoginView ID="LoginScreen" runat="server">
2         <AnonymousTemplate>
3            You can only set up your account
4            when you are logged in.
5         </AnonymousTemplate>
6         <RoleGroups>
7            <asp:RoleGroup Roles="Registered">
8               <ContentTemplate>
9             <asp:Button ID="ContinueRegistered"
10              runat="server" Text="Continue"/>
11              </ContentTemplate>
12           </asp:RoleGroup>
13           <asp:RoleGroup Roles="Admin">
14              <ContentTemplate>
15            <h4>Location</h4>
16            <asp:DropDownList ID="LocationList"
17               runat="server"/>
```

```
18            <asp:Button ID="ContinueAdmin"
19              runat="server" Text="Continue"/>
20            </ContentTemplate>
21        </asp:RoleGroup>
22      </RoleGroups>
23    </asp:LoginView>
```

Listing 2.6: Correct way of retrieving controls in a `LoginView` with a `RoleGroup`

```
1  LoginView LoginScreen;
2
3  private void Page_Load(object sender, EventArgs e)
4  {
5     Request myRequest = getRequest();
6     if (myRequest.isAuthenticated() && getUser.isInRole("Admin")) {
7        DropDownList list = (DropDownList)
8         LoginScreen.FindControl("LocationList");
9        list.DataSource = ...;
10       list.DataBind();
11    }
12 }
```

Chapter 3

Object Collaborations

No runtime entity exists independently in software, whether it be an object, component, or function. These entities interact and collaborate with each other in structured ways to make a useful program. As programmers, we manipulate these collaborations by performing operations on these entities, such as invoking methods, passing in arguments, setting stateful fields, and sending or receiving data through a port.

As seen in Vignette 2.2, the programmer must work with several objects that interact together (the page, the request, and the controls) in order for his application to produce the desired behavior (only show the drop down list when the user is logged in). This was a fairly complex collaboration between objects, but even smaller collaborations, such as that between an object, a collection it is in, and an iterator over the collection, happen regularly in programs.

While objects frequently participate in collaborations, not every collaboration is valid. They are frequently constrained in some way. For example, a list may require that all objects which are added to it be in a particular state. It is possible that the list checks this requirement, or perhaps that the item itself does, but it is also possible that the list assumes that the caller is responsible for enforcing this constraint. Therefore, the programmer must always be aware of which constraints she must abide by. I will refer to constraints on how several entities collaborate as *collaboration constraints*. Vignettes 2.1, 2.2 and 3.1 all contain examples of collaboration constraints.

Collaboration constraints occur with high frequency in software frameworks and are therefore of particular concern to that category of software. As we saw in Chapter 2, frameworks emphasize versatility and utility. In order for a framework to be highly versatile, it must provide mechanisms for the plugin to manipulate the internal representations of the framework and change how these objects collaborate. At every point where a framework opens this internal representation, there are implicit constraints on how the plugin may manipulate the collaboration. When a framework also aims to increase utility, it repeats this pattern many times, in overlapping areas of the API. This makes the implicit collaboration constraints even more confusing for a plugin developer.

It is the goal of this thesis to make collaboration constraints explicit so that they can be statically verified with a program analysis. This chapter addresses two sub-goals. First, I show that collaboration constraints are burdensome on plugin developers and therefore are worthy of further study. Second, I identify four important properties of collaboration constraints which any

21

(a) ASP.NET   **(b)** Spring
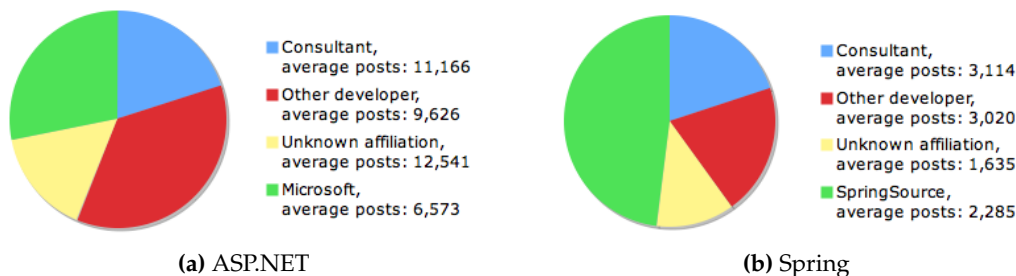
Figure 3.1: Corporate affiliations of the top 25 members of the Spring and ASP.NET help forums. Data gathered on April 12, 2011. Corporate affiliations determined by users self-description of their company and position. In both cases, the developers from Microsoft or SpringSouce were clearly labeled. Consultants are members who made it clear that their primary source of income was in consulting for use of the framework; most had books, blogs, and speaking arrangements listed. Other developers are people who use the framework as part of a job in another company. Unknowns are likely also other developers who chose to keep their affiliation private.

solution must address. This chapter will use an archival analysis of the ASP.NET help forums to address both of these goals.[1]

3.1   Why examine forums?

We can directly observe how difficult it is to use frameworks by inspecting posts on developer help forums, such as those for ASP.NET and Spring. When a developer chooses to post on a help forum, it tells us several things about his current situation:

- The developer has probably spent several hours trying to figure out the problem himself by searching for tutorials and documentation.

- The developer has probably asked his colleagues, who also did not know how to fix the problem.

- The developer has decided that it would be more efficient for him to anonymize the code, post it, and wait possibly several days for a response, rather than continue to puzzle it out alone.

The developers who respond to these posts are either more advanced developers or consultants and employees of companies that will benefit from others using this framework successfully. For example, some Microsoft teams require that employees spend several hours each month answering developer questions on the help forums. Many consultants also answer questions on the forums in hopes of selling their own third-party products or finding new clients. Figure 3.1 describes the affiliations of the top 25 posters on the ASP.NET and Spring help forums; notice that most of them are answering questions on the forums for indirect financial gain. The number of

---

[1]This analysis also lead to the discovery of Vignettes 2.1 and 2.2.
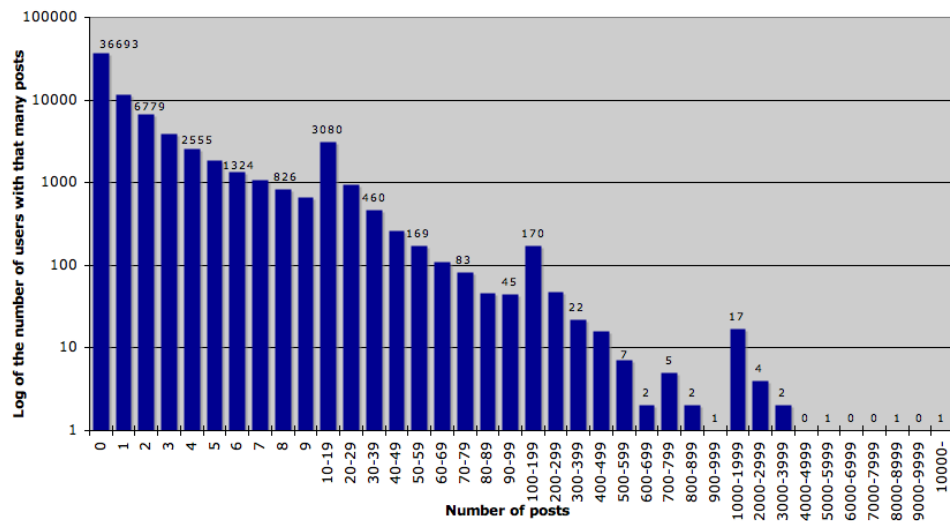
Figure 3.2: Post counts on the Spring web forums. The y-axis shows the number of users on a log scale. The x-axis shows the post count bucketed on a log scale. As seen, there were 36,693 registered users that had not posted at all; many of these appear to be failed spam-bots. Even so, there were over 11,000 users who made one post. Only 26 users made over a thousand posts, the highest post count was 10,275 posts by a single user. As can be seen, the regression is linear on a log-log scale. The vast majority of users post very few times.

posts per user is also exceedingly skewed for both forums; as seen in Figure 3.2, a very few users are doing most of the posting (presumably answering questions rather than asking them). Most people who post questions are fairly new to the forum (though not necessarily to the framework), and the questions are answered by a handful of experts who have limited time.

3.2 ASP.NET Forum Study

To further understand the type of questions people ask, I performed an archival analysis of the postings in the Web Controls sub-forum of the ASP.NET help forums. At the time of the analysis (spring of 2007), this was the most popular of the 104 sub-forums, with over 87,000 conversation threads since 2003. My analysis was on the threads that had their last activity during the first week of October in 2006. As the analysis itself was conducted many months later, each of these threads can be considered closed (that is, we expect no further helpful responses).

There were 271 threads with their last activity during this period. I first removed any threads which met one of the following properties:

- The question was not about Web Controls.

- The poster or responder used extremely poor English, to the point of not being understandable.

- The poster needed compilation help or otherwise did not understand basic syntax.

- The poster described the problem in such a vague way that it could not be reconstructed.

- There was no response at all or no response that solved the problem.

This left 66 threads which were on topic and were understandable enough to answer. Of these, 50 were requests for tutorials or documentation for a specific task.[2] This left 16 threads for study, which I have archived [2].

The remaining 16 threads had several interesting characteristics.[3] They were initiated by developers who had a problem in their code and were asking for help identifying the cause of the error and how to fix it. In these threads, the original posters provided their failing code and a detailed description of the failure, and a responding poster provided the fix and a description of why the code failed. Finally, each of these 16 threads (listed in Table 3.1a) described a problem where the developer was manipulating 2-5 objects within a collaboration and had broken a collaboration constraint.

These 16 threads show significant burden on the part of both the plugin and framework developer in several ways.

- As seen in Table 3.1a, only 7 of the faults resulted in a runtime exception; the remaining 9 resulted in incorrect behavior at run time, which is more difficult to debug than an exception with a message and a stack trace.

- Four of the faults were not local to the runtime error: based on the runtime error, the plugin developer would *not* be led to the method within their code that contained the fault, much less the line of code that contained it.

- There are three groups of threads, identified in the footnotes of the table, that are actually related issues that were posted about within the same week. It turns out that several of the constraints can fail in different ways at run time, depending on how they were broken, which makes it difficult for developers to search for other people who had a similar problem. As seen, even in a one week period, there were four threads discussing the issue shown in Vignette 2.1 and another four about a related lifecycle constraint.

- In two cases, a second poster appeared years later, after my initial study was finished. In both cases, the second poster came on simply to say that they had the same problem and that a search led them to this very helpful thread. One person noted that this saved hours of frustration, and another had already spent many hours trying to find an answer. This implies that developers will indeed search for a solution first and only post when a search turns up no useful answers. There are likely more developers who found these posts helpful and did not post in this manner.

---

[2]These posts would be ideally solved with design fragments [36] or similar techniques.

[3]I do not claim that those were the only 16, it is possible that I missed threads, that my knowledge of ASP.NET was not sufficient to understand the problem or solution being discussed, or that people continued to respond to posts much later (though I attempted to mitigate this issue by reading posts that were already several months old). I only claim that there were at least 16 out of the 271 which had these properties.

Number[*]	Runtime error	Runtime local?	#Posters	#Responders	Response time (H:MM)
1031123	Exception	No	1	1	3:23
1031139[†]	Exception	Yes	1	1	0:47
1031804	Incorrect Behavior	Yes	1	1	9:13
1032020[†]	Exception	Yes	1	0[**]	24:44 (over 1 day)
1031933	Incorrect Behavior	No	1	1	12:44
1030504[‡]	Incorrect Behavior	Yes	1	3	162:10 (over 6 days)
1027694[‡]	Incorrect Behavior	No	1	1	381:39 (over 15 days!)
1032187[‡]	Incorrect Behavior	Yes	2[††]	1[¶]	18:36
1032278	Exception	Yes	1	1	16:18
1032624[†]	Exception	Yes	2[‡‡]	1	2:10
1032991[§]	Exception	Yes	1	2	7:43
1033020[§]	Incorrect Behavior	Yes	1	2	3:02
1033046[‡]	Incorrect Behavior	Yes	1	1[¶]	1:46
1031946[§]	Exception	Yes	1	3	117:21 (over 4 days)
1033217[§]	Incorrect Behavior	No	1	2	3:13
1033450	Incorrect Behavior	Yes	1	1	260:22 (over 10 days)

(a) Properties that show burden

Number[*]	#Classes, #Objects	Extrinsic v. Instrinsic	Semantics	Artifact Types
1031123	3, 3	Intrinsic	Temporal, Identity	ASPX, VB.NET
1031139[†]	4, 4	Extrinsic	Identity, Value	ASPX, VB.NET
1031804	3, 2	Extrinsic	Value, Temporal, Identity	C#
1032020[†]	3, 3	Intrinsic	Identity, Value	ASPX, VB.NET
1031933	5, 5	Extrinsic	Callback, Identity	C#
1030504[‡]	4, 4	Extrinsic	Callback	C#
1027694[‡]	3, 2	Extrinsic	Callback	ASPX, C#
1032187[‡]	3, 3	Extrinsic	Callback	ASCX, VB.NET
1032278	4, 4	Extrinsic	Temporal, Identity	VB.NET
1032624[†]	4, 4	Intrinsic	Identity, Value	ASPX, C#
1032991[§]	2, 2	Extrinsic	Identity, Callback	C#
1033030[§]	2, 2	Extrinsic	Value, Callback	VB.NET
1033046[‡]	2, 2	Extrinsic	Callback	C#
1031946[§]	2, 2	Extrinsic	Callback	ASPX, C#
1033217[§]	2, 2	Extrinsic	Value, Callback	VB.NET
1033450	2, 2	Extrinsic	Value, Temporal, Identity	ASPX, VB.NET

[*] URL is `http://forums.asp.net/t/NUMBER.aspx`
[†] Related threads regarding proper usage of the `FindControl` method.
[‡] Related threads regarding when to dynamically create controls in the Page lifecycle.
[§] Related threads regarding when to access a field in the Page lifecycle.
[¶] None of the responders actually gave the correct response.
[**] Poster ended up "answering" own question, but actually got it slightly wrong.
[††] This thread had an additional responder after I concluded the study, written on November 24, 2010. The contents were "I must have read 10 or more post on how to do this but they were all so complicated I spent hours trying to understand one of them. Yours was great, I figured it out in a few minutes. Thank you for simplest example possible."
[‡‡] And another one on September 21, 2010! "this is precious...i did not know that...Perfect..saved me a lot of frustration :)"

(b) Properties of the underlying collaboration constraint

Table 3.1: Archival analysis of ASP.NET forum postings. These postings were understandable, solvable, on topic, and were not requests for a tutorial.

- The average time, from original posting to answer, was over 64 hours (about 2.67 days). Clearly, this is an inefficient way to debug a problem, which implies that most developers will use this as a method of last resort.

Based on this evidence, collaboration constraints appear to be burdensome for developers. While these problems were not the largest class of questions posted on the forum, they certainly required more time from developers in advance to investigate, and they require more time for the experts to read, understand, and answer as experts cannot simply point developers to an online tutorial or API. A solution which prevents these questions from being asked would not only free up time for the plugin developers but for the framework developers as well.

Plugin Vignette 3.1: Drop Down List

This example is from personal experience, rather than from the ASP.NET forums. I ran across this problem as a consulting developer at LEVEL Studios. We had a web page which had several, dynamically generated drop down lists on it. As they were dynamically generated, they were not in the ASPX file and were declared entirely in C#. The drop down lists were also paired; selecting an item in the first caused the second to be filtered. Selecting an item in the second caused an item in the first to be automatically selected.
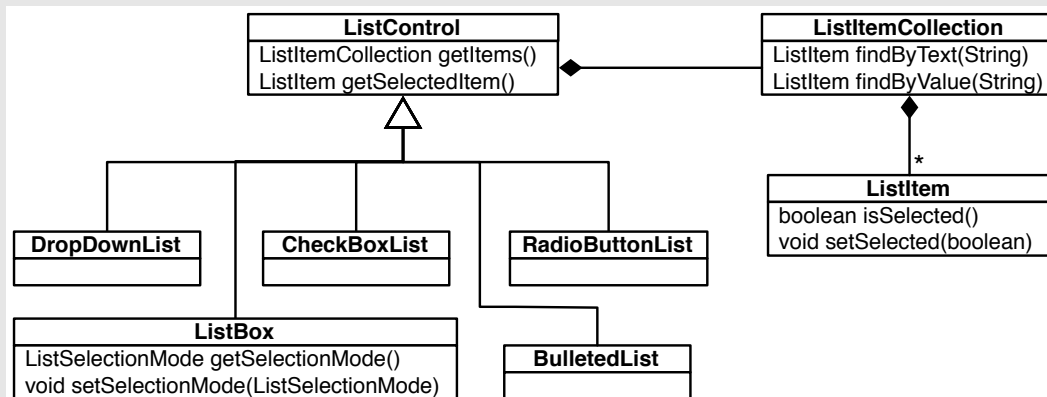
Figure 3.3: ASP.NET ListControl Class Diagram

The ASP.NET framework provides the relevant classes and methods to change the selection of a drop down list, as shown in Figure 3.3.[4] Notice that if the developer wants to change the selection of a `DropDownList` (or any other derived `ListControl`), she has to access the individual `ListItems` through the `ListItemCollection` and change the selection using `setSelected`. Based on this information, she might naïvely change the selection as shown in Listing 3.1. Her expectation is that the framework will see that she has selected a new item and will change the selection accordingly.

Listing 3.1: Incorrect selection for a `DropDownList`

```
1  DropDownList list;
2
3  private void Page_Load(object sender, EventArgs e)
4  {
```

```
5     ListItem newSel;
6     newSel = list.getItems().findByValue("foo");
7     newSel.setSelected(true);
8   }
```

When the developer runs this code, she will get the exception shown in Figure 3.4. The error message clearly describes the problem; a `DropDownList` had more than one item selected. This error is due to the fact that the developer did not de-select the previously selected item, and, by design, the framework does not do this automatically. While an experienced developer will realize that this was the problem, an inexperienced developer might be confused because she did not select multiple items.

Figure 3.4: Error with partial stack trace from ASP.NET

The stack trace in Figure 3.4 is even more interesting because it does not point to the code where the developer made the selection. In fact, the entire stack trace is from framework code; there is no plugin code referenced at all! At run time, the framework called the plugin developer's code in Listing 3.1, this code ran and returned to the framework, and then the framework discovered the error just before rendering the `DropDownList` into HTML. To make matters worse, the program control could go back and forth several times before finally reaching the check that triggered the exception. Since the developer doesn't know exactly where the problem occurred, or even what object it occurred on, she must search her code by hand to find the erroneous selection.

The correct code for this task is in Listing 3.2. In this code snippet, the developer de-selects the currently selected item before selecting a new item.

Listing 3.2: Correctly changing the selection

```
1   DropDownList list;
2
3   private void Page_Load(object sender, EventArgs e)
4   {
5     ListItem newSel, oldSel;
6     oldSel = list.getSelectedItem();
7     oldSel.setSelected(false);
8     newSel = list.getItems().findByValue("foo");
9     newSel.setSelected(true);
10  }
```

Now, as it turns out, I was quite familiar with this interesting aspect of the API. So, when I accidently wrote the code in Listing 3.3, I received the expected runtime error. Oops, I got the old item but I forgot to deselect it. Minor mistake, so I went back and edited the code to be like Listing 3.4. Notice, the only change is the addition of line 15. I then ran it, put it through various tests, and committed. Everything worked the way I expected.

Listing 3.3: Original bad code for manipulating selection of a `DropDownList`

```
1   private void Second_Selected(object sender, EventArgs e)
2   {
3      ListItem oldItem, newItem;
4      DropDownList firstList = ...
5      DropDownList secondList = ...
6      string newText;
7
8      oldItem = firstList.getSelectedItem();
9
10     //some code here that worked with oldItem
11     newText = //retrieve the new
12
13     newItem = firstList.getItems().findByText("foo");
14     newItem.setSelected(true);
15     //oops, forgot to deselect
16  }
```

Listing 3.4: "Corrected" version

```
1   private void Second_Selected(object sender, EventArgs e)
2   {
3      ListItem oldItem, newItem;
4      DropDownList firstList = ...
5      DropDownList secondList = ...
6      string newText;
7
8      oldItem = firstList.getSelectedItem();
9
10     //some code here that worked with oldItem
11     newText = //retrieve the new
12
13     newItem = firstList.getItems().findByText("foo");
14     newItem.setSelected(true);
15     oldItem.setSelected(false);
16  }
```

A couple of days later, the tester called me over with a very strange bug in my code. It turns out, I had missed an interesting case: if `newItem` happens to be the same as `oldItem`, then the item is selected (which does nothing, as it is already selected), and then it is de-selected. This leaves no items selected in the `DropDownList`, so the framework selects the first item in the list![5] This is an interesting issue, as it means that `ListItem.setSelected(false)` *must* occur before `ListItem.setSelected(true)`, and this is not a very obvious aspect of this constraint.

Listing 3.5: Using two `DropDownLists` together and using the wrong one

```
1   private void Second_Selected(object sender, EventArgs e)
2   {
3      ListItem oldItem, newItem;
4      DropDownList firstList = ...
5      DropDownList secondList = ...
```

```
 6      string newText;
 7
 8      oldItem = firstList.getSelectedItem();
 9      oldItem.setSelected(false);
10      //some code here that worked with oldItem
11      newText = //retrieve the new
12      newItem = secondList.getItems().findByText(newText);
13      newItem.setSelected(true);
14  }
```

Listing 3.6: Swapping the selection

```
 1  DropDownList list;
 2
 3  private void Page_Load(object sender, EventArgs e)
 4  {
 5     ListItem newSel, oldSel;
 6     newSel = list.getItems().findByValue("foo");
 7     newSel.setSelected(true);
 8     oldSel = list.getSelectedItem();
 9     oldSel.setSelected(false);
10  }
```

There are several other ways to break this constraint in seemingly correct ways. For example, Listing 3.5 deals with two `DropDownLists` where the developer accidentally uses the wrong list. Another way to break it would be to completely swap the method calls, as in Listing 3.6. Notice that the Line 8 *must* happen before Line 7. Otherwise, there is more than one item selected, and the call to at Line 8 may return the new `ListItem` rather than the old one, thus nullifying all of our changes!

There is another interesting aspects of this constraint in that the other sub-types of `ListControl` do not necessarily have this constraint. `RadioButtonList` has a similar constraint, but `CheckBoxList` can have as many or as few items selected as it likes. A `ListBox` is also interesting, as there is an setting to determine whether it will function as a single-select list or a multi-select list. Of course, the methods involved in the selection constraint are not in any of these subtypes, but in `ListControl` and `ListItem`.

Notice that this means that a `DropDownList` is *not* substitutable anywhere a `ListControl` is used! It has added an additional constraint which the parent did not have, and so it has broken behavioral subtyping. While unfortunate, this is not an uncommon problem in frameworks. The framework developers here have traded off usability of the external API for code reuse within the framework. They may have made the right choice (who would ever substitute a `DropDownList` for a multi-select `ListBox`?), but it has some unfortunate usability consequences.

---

[4]To make this code more accessible to those unfamiliar with C#, we are using traditional getter/setter syntax rather than properties.

[5]Clearly, I had a very good tester, as this problem only manifests when `oldItem` equals `newItem` *a*nd they are not the first item in the list.

3.3   Properties of Collaboration Constraints

Using these 16 threads in Table 3.1b as examples, I sought to understand the properties of collaboration constraints that make them difficult to specify using existing techniques such as typestate [16], pluggable typesystems [8], JML [66], or SCL [55]. I found four properties, as listed in Table 3.1b. In this section, I will also refer to Vignettes 2.1, 2.2, and 3.1 as examples of the properties.

Problem Property 1. Collaboration constraints involve multiple types and objects.

All of the problems described in the threads were examples of broken collaborations between several objects. Typically 2-5 objects were relevant for the collaboration, and 2-5 classes were also used (including relevant base classes). In Vignette 3.1, Listing 3.2 required four objects to make the proper selection. The framework code that the DropDownList example used was located in four classes (`DropDownList`, `ListControl`, `ListItemCollection`, and `ListItem`). In Vignette 2.2, the correct plugin also referenced four objects: the `Request` object, the `LoginView` control, the `DropDownList` control, and the `Page` in which all this code was running and which owned the `Request` and the `LoginView`.

Problem Property 2. Collaboration constraints are often extrinsic to a type.

Thirteen of the examples in Table 3.1b are *extrinisic* constraints, that is, the constraint is defined or checked outside of the type that is being constrained. By contrast, an *intrinsic* constraint is one which limits the class it is defined by; class invariants and single-object protocols are examples of intrinsic constraints. Vignette 3.1 provided an example of an extrinsic constraint. While the `DropDownList` was the class that checked the constraint (as seen by the stack trace), the constraint itself was on the methods of `ListItem`. However, the `ListItem` class is not aware of the `DropDownList` class or even that it is within a `ListControl` at all, and therefore it should not be responsible for enforcing the constraint. Likewise, in Vignette 2.1, the ability to call certain methods on a `Control` is limited based on what callback the `Page` is currently in, and not on any property of the `Control` itself.

Problem Property 3. Collaboration constraints involve semantic properties such as object identity, primitive values, state, and ordering of operations.

All of the examples in Table 3.1b required that the plugin developer be aware of the framework's program semantics in a way that goes beyond what is verifiable with traditional typesystems or structural checkers. In particular:

- *Object identity matters.* Nine constraints required developers to be aware not only of the type of the object, but the unique identity of the object. In Vignette 3.1, the plugin developer had to be aware of which `ListItem` she was using to avoid the problem in Listing 3.5. Likewise, in Vignette 2.2, the plugin developer had to use not just any `Request` object; he had to use the one which was associated with the `Page` that the `LoginView` was on.

- *Temporal requirements matter.* Four of the constraints had temporal requirements about the ordering of operations. As seen in Vignette 3.1, Listing 3.6, swapping two otherwise valid method calls can impact the collaboration in unexpected ways.

- *Primitive values matter.* Seven constraints referenced primitive values such as booleans and strings, and in some cases, the value directed control flow in the form of a dynamic state test. One example of this can be seen in Vignette 2.2, Listing 2.4, where it was not only important that the call be made to `Request.isAuthenticated()`, but that this call return `true`.

- *Callbacks matter.* Nine of the constraints were regarding a callback and specifically allowed or disallowed particular operations only within a particular method of the `this` object. Vignette 2.1 was entirely regarding a callback situation where certain operations were not allowed within certain contexts.

Every problem examined had at least one of these semantic issues, and 11 of them had at least two of these properties.

Problem Property 4. Collaboration constraints span many kinds of files and data, including declarative artifacts.

The examples studied spanned many different kinds of program artifacts, not just traditional program code. In particular, half of the studied examples were using a declarative artifact (either ASPX or ASCX) that was relevant to the constraint. Vignette 2.2 shows how a collaboration constraint extends into ASPX. In this example, the ASPX file affected how the programmer could reference and use the objects in the C# code-behind file. The code-behind file also had to use the same strings as the ASPX file for the desired behavior to take place. Vignette 2.1 contains another interesting interaction between these files. The field `DateYear` was not available because the framework uses dependency injection to automatically set this field for the plugin. Had the plugin set this field itself, the constraint no longer applies. Whether or not the framework performs the dependency injection in the code-behind file is based on what controls are declared in the ASPX file.

As discussed in related work, while there exist systems which can specify constraints with some of these properties, there are no known specifications systems that can specify all of them. In fact, very few existing systems can handle multiple objects in a generic way, very few can handle extrinsic constraints, and there is only one other piece of work known to do verification of declarative artifacts. Even the category of semantic properties, which is probably the one that is most frequently handled in related work, is not frequently done in a generic way. An in-depth discussion on how related work addresses these properties can be found in Chapter 8. The work presented in this dissertation aims to specify and analyze constraints with all four of these properties and do so in a cost-effective manner.

Chapter 4

Relationship Specifications

In Chapter 3, I described a collaboration constraint informally as a constraint on how several objects may interact in a protocol, and I used an archival analysis of the ASP.NET help forums to understand the properties of these constraints. I will now refine the definition more precisely below.

Definition 4 (Collaboration Constraint). A collaboration constraint is a precondition that is expressed as a predicate on the abstract states of several objects.

For example, there were two collaboration constraints in the problem in Vignette 3.1. The first was a precondition on calls to `ListItem.setSelected(boolean)` which said that when the operation is called on a `ListItem` that is a member of a `DropDownList` and the parameter is `true`, then the `DropDownList` must be in an unselected state. The second constraint, on the same operation, governed the case where the parameter is `false` and required that the `ListItem` be in the selected state. These states are abstract because they did not refer to a concrete memory representation of these two objects, and they did not describe how we know that the `ListItem` is connected to a `DropDownList` in memory. Notice that by combining several collaboration constraints together, a developer can describe a protocol for using multiple objects based on their abstract states.

Our goal is to provide a cost-effective specification and analysis technique for collaboration constraints that can handle all the properties described in Chapter 3. To achieve this, I will use abstract *relationships* between objects, defined below, as the primary abstraction for specifying and analyzing these constraints.

Definition 5 (Relationship). A relationship is a user-defined, abstract state-based association among several objects.

A relationship is a developer-defined abstraction that describes how several objects are associated at a design level. For example, we can talk about the relationship between a data structure and each item within that data structure. The actual code level association between two such objects may go through several other objects in the heap; a linked list, for example, might be associated with its tail-most object only through the pointers that go through every other object in

Listing 4.1: The definition of the Child relation. Every relation must define `params`, `effect`, and `test`

```
1  @Relation({ListItem.class, ListControl.class})
2  public @interface Child {
3      String[] params();
4      Effect effect();
5      String test() default "";
6  }
```

the list. However, we as programmers still talk about this association between the tail item and the list as though it is directly embedded in the code. A relationship is therefore a form of design intent and represents an abstract connection between several objects, not a concrete connection. It will be formally defined as a programmer-named predicate across runtime objects ℓ.

$$\mathrm{Relationship} = \mathrm{Name}(\ell_1, \ldots, \ell_n)$$

In Vignette 3.1, there was an association between `DropDownList` and each of the `ListItems`. This could be encoded as two relationships: Child(oldItem, ctrl) and Child(newItem, ctrl).

As relationships are entirely abstract and programmer defined, they can be thought of as an uninterpreted predicate. The static analysis described later will have no notion of the tacit meaning of a relationship and no way to check that it actually holds in code. In particular, relationships can represent ownership of objects (like Child(oldItem, ctrl)), but they are not interpreted as such and can hold whatever meaning the developer imposes on them.

While the relationship abstraction is not specific to a particular programming language[1], the specification language Fusion (Framework Usage SpecificatIONs) is implemented for Java and XML. To use the Child relationships above, we must first define the *relation* which describes the type of the relationship. In Fusion, this will be done with Java annotations; Listing 4.1 defines the Child relation. All relations are a Java annotation and are identified with the @Relation annotation that defines the types of the objects in the relationships. Additionally, all relations must have the three properties shown. In the rest of this chapter, we will assume that all relationships used have a relation defined in a similar manner.

In this chapter, I will show how Fusion can specify collaboration constraints by joining relationships with logical connectives to create preconditions on framework operations. I will also show how an associated static analysis can detect invalid plugins that do not meet these specifications. This chapter will show how to do this using the collaboration constraints from Vignettes 3.1 and 2.1 as concrete examples. Descriptions of how the analyses is affected by aliasing and how it works in the presence of declarative artifacts will be saved for Chapter 5.

---

[1]This dissertation does assume an imperative, object-oriented language. However, the "object" ℓ used in a relationship need not be an object as defined by the OO paradigm; many possible data abstractions could work here. For consistency though, I will use the term "object" throughout the dissertation.

Listing 4.2: Relationship effects on `List`

```
1  public interface List {
2      @Item({item, target}, ADD)
3      public void add(Object item);
4
5      @Item({item, target}, REMOVE)
6      public void remove(Object item);
7
8      @Item({item, target}, TEST, result)
9      public boolean contains(Object item);
10
11     @Item({∗, target}, REMOVE)
12     public void clear();
13 }
```

4.1 Specifying constraints in Fusion

Fusion uses relationships to define pre- and post-conditions of framework operations. A post-condition is described by *relationship effects*, and a pre-condition is described by a *requires predicate*. Unlike other pre- and post-condition specification systems, Fusion allows specifications to be written on many kinds of framework operations, not just method calls. The implementation currently supports method calls, constructor calls, the beginning of a method, and the end of a method. Theoretically, it can also support operations like field reads, field writes, and synchronizing on an object, though those operations are not implemented currently in Fusion. For purposes of describing the specifications, this section will primarily use method calls as the operation being specified.

Relationship effects are a type of post-condition that specifies the tacit knowledge of the plugin after calling a framework method. Consider a framework developer who is specifying a typical `List` interface where objects in the list are expected to be in an Item relationship with the list. The framework developer can specify that the method `List.add(Object item)` has the effect of creating an Item relationship between the item and the list (also known as the target object). Similarly, calling `List.remove(Object item)` removes the Item relationship between the item and the target object. The plugin can even test the state of this relationship by calling `List.contains(Object item)` to determine whether there exists an Item relationship between these objects.

Once the developer has defined a relationship type in Fusion, she can annotate methods to show relationship effects. Listing 4.2 shows the relationship effects for the simple `List` example.[2] To add or remove a relationship, the developer specifies the objects within the relationship (the `value` parameter in Listing 4.1) and the effect desired (the `effect` parameter in Listing 4.1). To test the state of a relationship, the developer uses the TEST effect and provides a value for the third parameter. This must be a boolean value which is true if the effect is added and false if it is removed.

---

[2]The syntax shown is not technically correct Java annotation syntax, but is shown this way for readability purposes. The correct syntax for @Item({item, target}, TEST, result) is actually @Item(value={"item", "target"}, effect=TEST, test="result")

Notice that a relationship effect only describes what is learned as a result of calling the method and does not necessarily reflect a change in the heap. For example, `List.contains` in Listing 4.2 is specified as either adding or removing an Item relationship, but the method of course is only a lookup and doesn't change the heap. The relationship, in some sense, already existed; the specification just provided us with belated information about it. In a similar manner, `ListControl.getSelectedItem` in Listing 4.3 is specified as adding two relationships, but of course, the getter does not change the heap. This reemphasizes that relationships are merely a developer abstraction about design intent; they have no direct correspondence to the code being specified. We will later see that this gives them a lot of power to verify APIs unidirectionally; they can be used to verify plugins but not to verify frameworks.

Relationship effects may refer to any variables used by the specified operation. In the case of method calls, relationships can refer to the parameters, the target of the method call or field access (designated with the name target), and the returned object (designated with result). Relationship effects may also refer to types and primitive values. Finally, parameters can be wild-carded, so Item({*, list}, REMOVE) removes *all* the Item relationships between list and any other object; this is especially useful to place on methods such as `List.clear()`, as shown in Listing 4.2. An example of these relationship effects on the `ListControl` API can be found in Listing 4.3; this API uses all three of the effects described and uses wildcards.

A pre-condition in Fusion is called a requires predicate; this is a logical predicate on relationships. The logical operators and, or, and implies are all allowed in a requires predicate, and relationships may be tested for falsehood using not (!). This allows the framework developer to write constraints such as "the item to deselect must already be selected and must be a member of the same drop down list as the item to be selected":

$$\mathsf{Selected(oldItem)} \wedge \mathsf{Child(oldItem,\ ctrl)} \wedge \mathsf{Child(newItem,\ ctrl)}$$

With just relationship effects and requires predicates, a framework developer could make simple pre- and post-conditions on operators. However, as described by Property 3, collaboration constraints have separate properties that are not capturable through this alone. Consider: how can we specify `ListItem.setSelected(boolean)` such that:

- We only deselect a `ListItem` that is currently selected.

- We only select a `ListItem` after deselecting.

- These operations are only allowed when the `ListItems` are members of the same `List-Control`.

- These operations are only constrained when the `ListControl` is a `DropDownList` or other single-select control.

To address this, Fusion provides a new kind of specification called a *trigger predicate*. While this predicate looks similar to the requires predicate, its meaning is very different. The trigger predicate determines *when* the constraint applies; it is more similar to the signature of the operator being constrained. While an operation's signature can syntactically describe when a constraint

Listing 4.3: Partial `ListControl` API with relationship effect annotations

```
1   public class ListControl {
2       @List({result, target}, ADD)
3       public ListItemCollection getItems();
4
5       //After this call we know two pieces of information. The returned item is selected and it is a child of this
6       @Child({result, target}, ADD)
7       @Selected({result}, ADD)
8       public ListItem getSelectedItem();
9   }
10  public class ListItem {
11      //If the return is true, then we know we have a selected item. If it is false, we know it was not selected.
12      @Selected({target}, TEST, return)
13      public boolean isSelected();
14
15      @Selected({target}, TEST, select)
16      public void setSelected(boolean select);
17
18      @Text({result, target}, ADD)
19      public String getText();
20
21      //When we call setText, remove any previous Text relationships, then add one for text
22      @Text({∗, target}, REMOVE)
23      @Text({text, target}, ADD)
24      public void setText(String text);
25  }
26  public class ListItemCollection {
27      @Item({item, target}, REMOVE)
28      public void remove(ListItem item);
29
30      @Item({item, target}, ADD)
31      public void add(ListItem item);
32
33      @Item({item, target}, TEST, result)
34      public boolean contains(ListItem item);
35
36      @Item({result, target}, ADD)
37      @Text({text, result}, ADD)
38      public ListItem findByText(String text);
39
40      //if we had any items before this, remove them after this call
41      @Item({∗, target}, REMOVE)
42      public void clear();
43  }
```

Listing 4.4: DropDownList Selection Constraints

```
1   @Constraint(
2       op="ListItem.setSelected(boolean select)",
3       trigger="select == false and Child(target, ctrl) and ctrl instanceof DropDownList",
4       requires="Selected(target)",
5       effect={"!CorrectlySelected(ctrl)"})
6   @Constraint(
7       op="ListItem.setSelected(boolean select)",
8       trigger="select == true and Child(target, ctrl) and ctrl instanceof DropDownList",
9       requires="!CorrectlySelected(ctrl)",
10      effect={"CorrectlySelected(ctrl)"})
11  @Constraint(
12      op="end−of−method",
13      trigger="ctrl instanceof DropDownList",
14      requires="CorrectlySelected(ctrl)",
15      effect={})
16  public class DropDownList {...}
```

should apply (ie: this is a constraint on `ListItem.setSelected(boolean)`), a trigger can semantically describe when a constraint should apply (ie: only when the `ListItem` is a member of a `DropDownList` and when the `boolean` parameter is false).

In Fusion, we can use trigger predicates with requires predicates and relationship effects to specify a constraint on an operation. This is done using a Java annotation with four parts.

1. *operation*: This is a signature of an operation to be constrained, such as a method call, constructor call, or even a tag signaling the end of a method. Notice that these constraints may be defined in another class. This makes constraints more expressive than a class or protocol invariant as they can be extrinsic.

2. *trigger predicate*: This is a logical predicate over relationships. The plugin's relationship context must determine that this predicate holds for this constraint to be triggered. If not, the constraint is ignored. While *operation* provides a syntactic trigger for the constraint, *trigger* provides the semantic trigger. The combination of both a syntactic and semantic trigger allows constraints to be more flexible and expressible than many existing protocol-based solutions.

3. *requires predicate*: This is another logical predicate over relationships. If the constraint 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. If the constraint is triggered, these effects will be applied in the same way as the relationship effects described earlier. They will be applied regardless of the state of the requires predicate.

Listing 4.4 provides the three Fusion constraint specifications needed to completely describe the collaboration constraint of Vignette 3.1, including a specification for each mode of `ListItem.`–

`setSelected(boolean)`. The first constraint is checking that at every call to `ListItem.setSelect‑ed(boolean)`, if the relationship context shows that the argument is false, the receiver is a Child of a `ListControl`, and if that `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 it enforces proper selection of `ListItems` in a `DropDownList`. The third constraint ensures that the plugin method does not end in an improper state by utilizing the "end-of-method" instruction to trigger when a method is about to end. This ensures that all `DropDownLists` are left in a state where only one item is selected.

4.2 Analyzing Programs

One of the primary benefits of formal specifications is using them to statically verify programs. Fusion provides a static analysis to track relationships through plugin code and check plugin code against framework constraints. The Fusion analysis is a modular, branch-sensitive, forward dataflow analysis[3]. It is designed to work on a three address code representation of Java-like source. The analysis runs in the Crystal static analysis framework, which provides all of these features. In this section, I will present the analysis data structures, the intuition behind the three variants of the analysis, and examples of how it works on the example in Vignette 3.1.

The Fusion analysis requires that it be provided with a points-to analysis that implements a simple interface. First, it assumes there is a context $\mathcal{L}$ that given any variable x, provides a finite set $\bar{\ell}$ of abstract locations that the variable might point to. Second, it assumes a finite 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 $\mathcal{A}$. This lattice must conservatively abstract the heap, as defined by Definition 6.

Definition 6 (Abstraction of Alias Lattice). Assume that a heap h is defined as a set of source variables x which point to a runtime location ℓ of type τ. Let H be all the possible heaps at a particular program point. An alias lattice $< \Gamma_\ell, \mathcal{L} >$ abstracts H at a program counter if and only if

$\forall\, h \in H \,.\, dom(h) = dom(\mathcal{L})$ and
$\quad \forall\, (x_1 \hookrightarrow \ell_1 : \tau_1) \in h \,.\, \forall\, (x_2 \hookrightarrow \ell_2 : \tau_2) \in h \,.$
$\quad\quad (\text{if } x_1 \neq x_2 \text{ and } \ell_1 = \ell_2 \text{ then}$
$\quad\quad\quad \exists\, \ell' \,.\, \ell' \in \mathcal{L}(x_1) \text{ and } \ell' \in \mathcal{L}(x_2) \text{ and } \tau_1 <: \Gamma_\ell(\ell')) \text{ and}$
$\quad\quad (\text{if } x_1 \neq x_2 \text{ and } \ell_1 \neq \ell_2 \text{ then}$
$\quad\quad\quad \exists\, \ell_1', \ell_2' \,.\, \ell_1' \in \mathcal{L}(x_1) \text{ and } \ell_2' \in \mathcal{L}(x_2) \text{ and } \ell_1' \neq \ell_2' \text{ and } \tau_1 <: \Gamma_\ell(\ell_1') \text{ and } \tau_2 <: \Gamma_\ell(\ell_2'))$

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 the two variables are not aliased within a given heap, 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.

---

[3]By branch-sensitive, we mean that the true and false branches of a conditional may receive different lattice information depending upon the condition. This is not a path-sensitive analysis.

More details on how Fusion uses a given points-to analysis can be found in Chapter 5, for now it is enough to know that it must meet the requirement above.

4.2.1   The Relation Lattice

The status of a relationship is tracked using the four-point dataflow lattice represented in Figure 4.1, where Unknown represents either True or False and the bottom of the lattice, $\perp$ is a special case used only inside the flow function. The Fusion 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 $\mathcal{A}$.
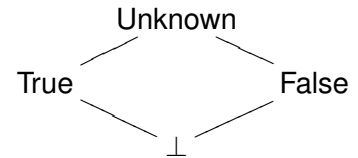
Figure 4.1: The relationship state lattice.

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 ℓ and a finite number of relationships. As the flow function is monotonic, the analysis always reaches a fix-point.

Since the relationships are tracked with three possible states, True, False, or Unknown, a relationship predicate like the trigger and requires predicates must be evaluated with three-value logic. The formal rules used to evaluate a relationship predicate under a given lattice is shown in Appendix B (Figures B.19, B.21, and B.22), but it follows the expected pattern of a three-value logic.

4.2.2   The 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 effects to the lattice. The flow function is defined as

$$f_{\mathcal{C}}(\mathcal{A}, \rho, \mathrm{instr}) = \rho'$$

where $\mathcal{C}$ is all the constraints, $\mathcal{A}$ is the alias lattice, ρ is the starting relationship lattice, ρ' is the ending relation lattice, and instr is the instruction the analysis is currently checking. 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. These bindings will be used for the remaining steps.

Once the analysis has a match, two things must occur. First, it uses the bindings generated above to show that the requires predicate of the constraint is true under ρ. If it is not true, then the analysis reports an error on instr. Second, the analysis must use the same bindings to produce ρ' by applying the relationship effects. If the analysis reports an error, then the flow function above will terminate with no result.

Listing 4.5: Walk-through showing how the lattice ρ changes as the analysis flows through the program.

```
1   DropDownList ddl = ...;
2   ListItemCollection coll;
3   ListItem newSel, oldSel;
4   //−
5   oldSel = ddl.getSelectedItem();
6   //Child(l₂, l₁), Selected(l₂)
7    oldSel.setSelected(false);
8   //Child(l₂, l₁), !Selected(l₂)
9   coll = ddl.getItems();
10  //Child(l₂, l₁), !Selected(l₂), List(l₃, l₁)
11  newSel = coll.findByText("foo");
12  //Child(l₂, l₁), !Selected(l₂), List(l₃, l₁), Item(l₄, l₃), Text("foo", l₄)
```

As an example for how this works, consider the code snippet in Listing 4.5. In this listing, the comments show the lattice ρ. At line 7, the starting lattice ρ is:

$$\text{Child}(\ell_2, \ell_1) \mapsto \text{True}$$
$$\text{Selected}(\ell_2) \mapsto \text{True}$$

All relationships that are not explicitly shown are assumed to be Unknown. The points-to lattice $\mathcal{A}$ is not shown in the listing, but for purposes of this example it might be given as:

$$\Gamma_\ell = \{\ell_1 : \texttt{DropDownList}, \ell_2 : \texttt{ListItem}\}$$
$$\mathcal{L} = \{\texttt{oldSel} = \{\ell_1\}, \texttt{ddl} = \{\ell_2\}\}$$

The analysis will then check every constraint to see if there is a matching operator and a matching trigger. It might eventually find the two constraints below:

```
1   @Constraint(
2       op="ListItem.setSelected(boolean select)",
3       trigger="select == false and Child(target, ctrl) and ctrl instanceof DropDownList",
4       requires="Selected(target)",
5       effect={"!CorrectlySelected(ctrl)"})
6   @Constraint(
7       op="ListItem.setSelected(boolean select)",
8       trigger="select == false",
9       requires="TRUE",
10      effect={"!Selected(target)"})
```

Therefore, as the operator matches and the trigger evaluates to True for both of these constraints, the analysis will produce the output lattice ρ′, which will be used as the input for the next line. When more than one constraint applies, the resulting effects are merged together to produce a single ρ′:

$$\text{Child}(\ell_2, \ell_1) \mapsto \text{True}$$
$$\text{Selected}(\ell_2) \mapsto \text{False}$$
$$\text{CorrectlySelected}(\ell_1) \mapsto \text{False}$$

The analysis is conservative in this merge but attempts to save as much precision as possible; Appendix B describes it in further detail. Any constraints where the operator does not match, or where the trigger evaluates to False, are ignored and their effects are not applied. In cases where the trigger evaluates to Unknown, all the relationships in the effects list are set to Unknown in order to be conservative. Again, the analysis does actually try to save some precision using further tricks, such as comparing to the old state, as explained in Appendix B.

4.2.3   Soundness and completeness

Soundness and completeness allow the user of the analysis to either have confidence that there are no errors at run time if the analysis finds none (if it is sound) or that any errors the analysis finds will actually occur in some run time 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 generate a new lattice for a particular instruction.

I define soundness and completeness of the Fusion analysis by assuming the existence of a points-to analysis which abstracts the heap using $\mathcal{A}$, as described above. For both of these theorems, let $\mathcal{A}^{conc}$ define the *actual heap* at some point of a real execution, and let $\mathcal{A}^{abs}$ be a sound approximation of $\mathcal{A}^{conc}$ by Definition 6. Additionally, 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 $\rho^{conc} \sqsubseteq \rho^{abs}$.

For the sound variant, we expect that if the flow function generates a new lattice using the imprecise lattice ρ^{abs}, then any more concrete lattice will also produce a new lattice for that instruction. As the flow function *only generates a new lattice 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 new abstract lattice must conservatively approximate any new concrete lattice. Theorem 1 captures the intuition of local soundness formally.

Theorem 1 (Local Soundness of Relations Analysis)**.**
$\quad$ if $f_{\mathbb{C},\mathcal{A}^{abs}}(\rho^{abs}, instr) = \rho^{abs'}$ and $\rho^{conc} \sqsubseteq \rho^{abs}$
$\quad$ then $f_{\mathbb{C},\mathcal{A}^{conc}}(\rho^{conc}, instr) = \rho^{conc'}$ and $\rho^{conc'} \sqsubseteq \rho^{abs'}$

If the Fusion analysis is complete, we expect a theorem which is the opposite of the soundness theorem and is shown in Theorem 2. If a flow function generates a new lattice given a lattice ρ^{conc}, then it will also generate a new lattice on any abstraction of ρ^{conc}. An analysis with this property may produce false negatives, as the analysis can find an error using the concrete lattice yet generate a new lattice using ρ^{abs}, but it will produce no false positives. Like the sound analysis, the resulting lattices must maintain their existing precision relationship.

Theorem 2 (Local Completeness of Relations Analysis)**.**
$\quad$ if $f_{\mathbb{C},\mathcal{A}^{conc}}(\rho^{conc}, instr) = \rho^{conc'}$ and $\rho^{conc} \sqsubseteq \rho^{abs}$
$\quad$ then $f_{\mathbb{C},\mathcal{A}^{abs}}(\rho^{abs}, instr) = \rho^{abs'}$ and $\rho^{conc'} \sqsubseteq \rho^{abs'}$

For this work, I have implemented both a sound variant and a complete variant of the Fusion analysis. Additionally, I have created a third variant, known as the *pragmatic variant*, which attempts to balance the tradeoffs between soundness and completeness. This variant is unique in

Table 4.1: Predicate checking differences between sound, complete, and pragmatic variants.

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

the research literature, but it could be created for other analyses with similar properties to Fusion. In particular, any system which has separate concepts of a trigger predicate and a requires predicate can support a pragmatic variant.

The formal semantics for the three variants can be found in Appendix B, and the proofs of the two theorems above, for the appropriate variants, can be found in Appendix C. Global soundness and global completeness directly follow from local soundness and local completeness due to the monotonicity of the flow function and the initial conditions of the lattice. Appendix C contains further discussion on how these global properties hold and why the analysis is monotonic; further reading on the theoretical properties of monotonic dataflow analyses can be found in [80].

The primary difference in the three variants is how they handle *unknownness* from the trigger and requires predicates. As stated before, the relationship lattice uses Unknown, in addition to True and False, which results in predicates that are evaluated as three-value logic. How the variants deal with Unknown in each of these predicates is defined in Table 4.1 and is described below.

Trigger condition. The trigger predicate determines when the constraint will check the requires predicate and when it will produce effects. The sound variant must trigger a constraint whenever there is even a possibility of it triggering at run time. 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. The pragmatic variant will work the same as the complete variant when determining whether to trigger the constraint. The rationale here is to try to reduce the number of false positives by only checking constraints when they are known to be applicable.

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 requires predicate is False or Unknown. The complete variant, however, can only cause an error if it is sure there is one, so it only flags an error if the requires predicate is definitely False. In this case, the pragmatic variant will work the same as the sound variant. If the analysis has come to this point, it already has enough information to determine that the trigger was true. Therefore, we will require that the plugin definitely show that the requires predicate is True, with the expectation that this will reduce the false negatives.

While the pragmatic variant can produce false positives and false negatives, it provides a unique point in the space. It takes advantage of the heuristic that if there is enough precision to tell whether the trigger predicate is True or False, then there ought to be enough precision to tell this for the requires predicate as well. Any other specification system which provides a separate concept for a trigger predicate can also create a pragmatic variant.

We shall now explore how the three variants compare on the examples from Vignette 3.1. Table 4.2 summarizes each of the snippets from Vignette 3.1, where the fault in the snippet is, and the

Table 4.2: Results from running each variant on the examples from Vignette 3.1.

Listing reference	Line number of fault	Sound results	Pragmatic results	Complete results
3.1: Naïve selection	7	7	7	-
3.2: Correct selection	-	9,9	-	-
3.3: Forgotten deselection	14	14, 14	14	-
3.4: Nothing selected	14	14	14	14
3.5: Two lists, incorrect	13	13, 13	13	-
(Not given): Two lists, correct	-	13, 13	-	-
3.6: Swapped selection	7,9	7, 9, 9	7, 9	9

All possible Trigger/Requires combinations
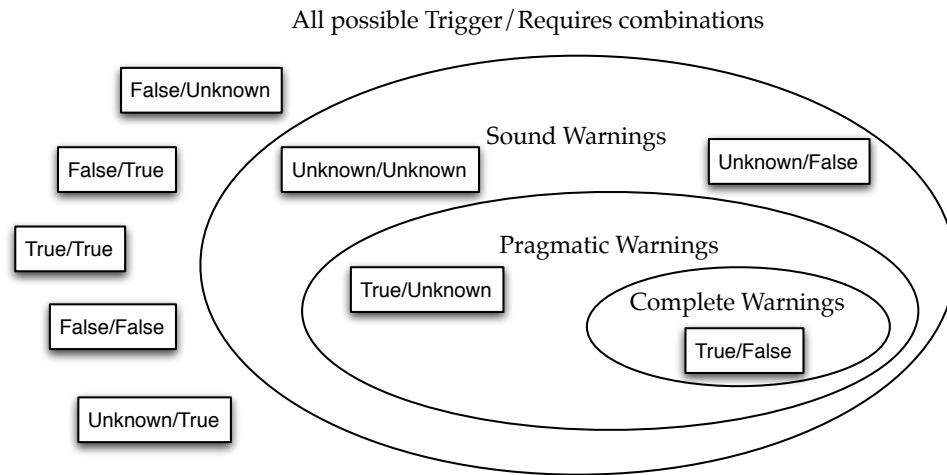
Figure 4.2: Venn diagram of warnings reported by each variant.

results from the three variants of the analysis when using the specifications from Listing 4.3 and Listing 4.4.

Notice that the results produced by the variants have a subset relationship. This is always the case; as seen in Figure 4.2, the variants are defined in such a way that the pragmatic variant will always contain the results of the complete variant, and it will attempt to take the parts of the complete variant that are heuristically more likely to be true positives than false positives.

Listing 4.6 and 4.7 show the snippet from the first two rows of Table 4.2 with the relationship lattice described in comments. For simplicity of the examples, the alias lattice is not shown and all variables are assumed to be unique in the example. Notice that for all variants, the relationship lattice is the same. This is because all three variants must be conservative when producing the relationship effects. Excluding the complexities with aliasing that we will see in Chapter 5, the *only* difference between the variants are the condition that lead to an error. As presented so far, the dataflow analysis works identically.

At line 9 in Listing 4.6, both the first and second constraints' operators will match the instruc-

Listing 4.6: Incorrectly changing the selection, with ρ in comments.

```
1   DropDownList list;
2
3   private void Page_Load(object sender, EventArgs e)
4   {
5       ListItem newSel;
6   //−
7       newSel = list.getItems().findByValue("foo");
8   //Child(newSel, list), Value("foo", newSel)
9       newSel.setSelected(true);
10  //Child(newSel, list), Value("foo", newSel), Selected(newSel)
11  }
```

Listing 4.7: Correctly changing the selection, with ρ in comments.

```
1   DropDownList list;
2
3   private void Page_Load(object sender, EventArgs e)
4   {
5       ListItem newSel, oldSel;
6   //−
7       oldSel = list.getSelectedItem();
8   //Child(oldSel, list), Selected(oldSel)
9       oldSel.setSelected(false);
10  //Child(oldSel, list), !Selected(oldSel), !CorrectlySelected(list)
11      newSel = list.getItems().findByValue("foo");
12  //Child(oldSel, list), !Selected(oldSel), !CorrectlySelected(list), Child(newSel, list), Value("foo", newSel)
13      newSel.setSelected(true);
14  //Child(oldSel, list), !Selected(oldSel), CorrectlySelected(list), Child(newSel, list), Value("foo", newSel), Selected(newSel)
15  }
```

tion signature. However, the first constraint's trigger predicate will evaluate to False, so it will be ignored as though the operator didn't match. The second constraint's trigger predicate evaluates to True, so all the variants will evaluate the requires predicate. As this will evaluate to Unknown, both the sound and pragmatic variants will produce an error at line 9. On the other hand, the complete variant does not have enough precision to discover the error.

Let's now consider the code in Listing 4.7. When Fusion analyzes line 9, it will again try both the first and second constraints in Listing 4.4. However, this time the first constraint's trigger predicate will evaluate to True and the second constraint's will be False. All the variants will therefore evaluate the required predicate of the first constraint. As this evaluates to True as well, all the variants will pass and apply the effects. The analysis works down to line 13, where the second constraint matches both the operator and the trigger predicate. As the requires predicate is True again, all variants should pass and apply effects.

The astute reader may have noticed a discrepancy: all the variants passed in Listing 4.7, yet Table 4.2 reports that the sound analysis produces two false positives. This is because the results shown in Table 4.2 are from running the sound analysis alongside a sound may-alias analysis,

Listing 4.8: A fourth constraint that improves the precision of the analyses.

```
1  @Constraint(
2      op="begin−of−method",
3      trigger="TRUE",
4      requires="TRUE",
5      effect={"CorrectlySelected(∗)"})
```

whereas in Listing 4.7, we assumed a must-alias analysis. As we will see in the next chapter, the results of all three variants are strongly tied to the points-to analysis.

The results are also strongly tied to the precision of the specifications. For example, adding the specification in Listing 4.8, which adds the relationship CorrectlySelected for all `DropDownLists` at the beginning of every method, will allow the complete variant to detect the bug in Listing 4.6. The full discussion of how specifications impact the precision of the analysis, including the impact of automatically generated specifications, can be found in Chapter 7.

4.3   Other kinds of specifications

In prior sections, I have used non-relationship predicates like "select == true" or "ctrl instanceof DropDownList" within a trigger or requires predicate. Both of these are "special purpose" relationships with a predefined semantics. This section will now describe how these are used and the analyses that are associated with them.

I previously introduced relationship effects and constraints as two kinds of specifications in Fusion. In this section, I will describe a third kind of specification specifically for callbacks. Both callbacks and relationships effects are syntactic sugar and can be converted into the basic @Constraint specification. Finally, this section will introduce inferred relationship specifications, which are uncommonly used but highly expressive feature of Fusion.

4.3.1   Special purpose relationships

While most of the relationships have an uninterpreted user-defined semantics, it is sometimes useful to have relationships with a little more power. Therefore, I have provided pre-defined semantics for the equality relation (==) and the type relation (instanceof) for usability purposes.

The Fusion analysis depends on other analyses to evaluate these predicates. The points-to analysis already used can evaluate both reference equality and type relationships. Additionally, Fusion uses a boolean constant propagation analysis to evaluate boolean variables and boolean equality, like "select == true". It is relatively straightforward to add these special-purpose analyses, and we can imagine extensions to handle integers, enums, and strings as well.

4.3.2   Converting relationship effects

Relationship effects are syntactic sugar that can be easily translated into a constraint form. Relationship effects are translated by considering them as a constraint on the annotated method with

```
public class ListControl {
    @Child({result, target}, ADD)
    @Selected({result}, ADD)
    public void getSelectedItem() {

        ...

    }
    ...
}

public class ListItem {
    @Selected({target}, TEST, sel)
    public void setSelected(boolean sel) {

        ...

    }
    ...
}
```

```
@Constraint(
 op = "ListControl.getSelectedItem()",
    trigger = "true",
    requires = "true",
    effect = {"Child(result, target)", "Selected(result)"})
public class ListControl {...}

@Constraint(
 op = "ListItem.setSelected(boolean sel)",
    trigger = "sel == TRUE",
    requires = "TRUE",
    effect = {"Selected(target)"})
@Constraint(
 op = "ListItem.setSelected(boolean sel)",
    trigger = "sel == FALSE",
    requires = "TRUE",
    effect = {"!Selected(target)"})
public class ListItem {...}
```

Figure 4.3: Translating relationship effects into constraints.

a True trigger predicate, a True requires predicate, and the effect list as annotated. Test effects are translated into two constraints which use boolean equality. Figure 4.3 shows example effects translated into constraints.

4.3.3 Callbacks

While relationship effects provide information to a caller, callback states provide information to a callee. When frameworks make callbacks into plugin code, there is an implicit contract regarding when the callback will occur and the states of objects at this point. For example, Vignette 2.1 showed the the plugin developer should be aware that the Page's controls do not exist in the PreInit callback and do not have data until the Load callback.

The framework developer can specify this using the @Callback annotation. The @Callback annotation takes the name of an unary relation on the type of the target object.[4] An example of using this specification can be seen in Listing 4.9. This example shows how the callback relationships can be used in the constraints to prevent calls from happening within certain callbacks or to only allow them to be used within some callbacks.

The @Callback annotations above are translated into constraints with an operation that matches a "beginning of method" tag on the specified method and a set of effects where the specified callback relationship is set to True and all others are set to False, as shown in Listing 4.10

With the specifications in Listing 4.9, the defect in Vignette 2.1 would be found by all three variants, as can be seen in Listing 4.11.

---

[4]The @Callback annotation is very similar to typestate, and indeed, typestate can be used instead of having a state declaration in this form.

Listing 4.9: Specifications for problem in Vignette 2.1.

```
 1  @Constraint(
 2    op="ListControl.∗∗",
 3    trigger = "SubControl(target, page)",
 4    requires = "!PreInit(page)",
 5    effects = {}
 6  )
 7  @Constraint(
 8    op="ListControl.setDataSource(List data)",
 9    trigger = "SubControl(target, page)",
10    requires = "Loaded(page)",
11    effects = {}
12  )
13  public class Page {
14      @Callback("PreInit")
15      protected void Page_PreInit(object sender, EventArgs e);
16
17      @Callback("Initialized")
18      protected void Page_Init(object sender, EventArgs e);
19
20      @Callback("Loaded")
21      protected void Page_Load(object sender, EventArgs e);
22  }
```

Listing 4.10: Translated callback specifications from Listing 4.9.

```
 1  @Constraint(
 2    op="BOM:Page.Page_PreInit(object sender, EventArgs e)",
 3    trigger = "TRUE",
 4    requires = "TRUE",
 5    effects = {PreInit(target), !Initialized(target), !Loaded(target)}
 6  )
 7  @Constraint(
 8    op="BOM:Page.Page_Init(object sender, EventArgs e)",
 9    trigger = "TRUE",
10    requires = "TRUE",
11    effects = {!PreInit(target), Initialized(target), !Loaded(target)}
12  )
13  @Constraint(
14    op="BOM:Page.Page_Load(object sender, EventArgs e)",
15    trigger = "TRUE",
16    requires = "TRUE",
17    effects = {!PreInit(target), !Initialized(target), Loaded(target)}
18  )
```

Listing 4.11: Incorrect usage of the page lifecycle with ρ in comments using the constraints from Listing 4.9.

```
1  DropDownList DateYear;
2
3  public Page_PreInit(object sender, EventArgs e)
4  {
5      List<DateTime> Dates;
6  //SubControl(this, DateYear), PreInit(this), !Initialized(this), !Loaded(this)
7      if (!isPostBack)
8          for (int i = 0; i < 4; i++)
9              Dates.Add(System.DateTime.Now.AddYears(i));
10
11  //SubControl(this, DateYear), PreInit(this), !Initialized(this), !Loaded(this)
12      DateYear.setDataSource(Dates);  //constraints will fail
13      DateYear.setDataTextField("Year");
14      DateYear.DataBind()
15  }
```

Listing 4.12: Using the Infer specifications to create effects

```
1  @Infer(
2      trigger="Item(item, list) and List(list, ctrl)",
3      effect={"Child(item, ctrl)"})
4  public class ListControl {...}
```

4.3.4 Inferred Relationships

In some cases, the relationships between objects is implicit. Consider the `ListItemColletion` from the `DropDownList` example. In this example, the framework developer would like to state that the items in this list are in a Child relationship with the `ListControl` parent. While we can annotate the `ListItemCollection` class with this information, it seems non-ideal as the `ListItemCollection` should not know about `ListControls`.

Inferred relationships describe these implicit relationships that can be assumed any time some other relationship predicate is true. Listing 4.12 shows an example for inferring a Child relationship based on the relations Item and List. Whenever the relationship context can show that the *trigger* predicate is true, it can infer the relationship effects in the *effect* list. Inferred relationships allow the framework developer to specify relationship effects that would otherwise have to be placed on every location that the predicate is true; this would significantly drive up the cost of adding these specifications. While the example in Listing 4.12 could have been written as a traditional constraint, inferred relationships are particularly useful in cases where we need closures to use the specification several times to create relationships within a more complex data structure, like a list with no predefined size. In practice, only one API from the Spring case study, discussed in Section 6.4.3, needed this specification form, but its expressive power made it worthy of inclusion

It is possible to produce inferred relationships that directly conflict with the relationship con-

text. 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. The rationale behind this is that the constraints and relationship effects are explicitly declared, and this should be reflected by the giving them precedence. An alternative mechanism would be to signal an error, though it is not currently clear whether this will increase the number of false positives.

Currently, these specifications are only used on an as-needed basis using backwards chaining. Because inferred relationships are not generated at every step of the analysis, this is an unsound and incomplete feature of Fusion, so it is only use by the pragmatic variants for now. This could be changed if Fusion used forward chaining to greedily create all possible relationships at each step in the analysis; such an analysis would preserve soundness and completeness, though it would be very expensive to run.

4.4 Achievement of solution goals

In earlier chapters, I set up several goals for a specification and analysis system. First, it must address the common properties of collaboration constraints, as given in Chapter 3. Second, it must be an adoptable, cost-effective solution, as proposed in Chapter 1. In this section, I will briefly evaluate Fusion against these goals.

4.4.1   Can Fusion capture collaboration constraints?

Property 1: Multiple objects   As a relationship captures the associations between several objects, it is a good representation for collaboration constraints. Relationships can also be used to "build-up" a collaboration in cases where not all the objects involved exist at the start of the collaboration; for example, the second `ListItem` in Vignette 3.1 does not appear until halfway through the collaboration.

Property 2: Extrinsic   Relationships are not owned by any particular type, so crossing a type boundary is not an obstacle. The constraints in Fusion can be used to constrain any visible type. Unlike other specifications, they are not restricted to the defining type. Additionally, the object being constrained might not be aware of the constraint itself, as the relationships can be added without its knowledge.

Property 3: Semantic issues   As seen, Fusion can handle a wide variety of semantic issues. The ability to specify callbacks is built into the language, and the ability to specify ordering of operations is made possible through the use of the trigger predicate to chain several constraints together. The trigger predicate also makes it possible to specify different constraints based upon primitive values or object identity.

Property 4: Many artifacts   Relationships are not a language-specific abstraction; they are a design abstraction. Any language with the concept of distinct entities and collaborations between

entities can use relationships to describe the collaborations. While this chapter did not show specific examples with declarative files, the next chapter will use Vignette 2.2 to show how they are handled.

4.4.2 Does Fusion meet the goals for an adoptable, cost-effective tool?

One of the stated goals of Fusion was to be an adoptable, cost-effective solution to the problem. Chapter 7 will discuss this in more detail. There are four primary reasons why I believe this to be an adoptable solution compared to existing work.

Minimize specification writing costs. The primary cost of using any specification and analysis system is the cost of writing specifications. It seems to defeat the purpose of such as system to require the plugin developer, who is already struggling to understand the framework, to also learn a new language and specify his code. Therefore, Fusion has specifications only on the framework, which can be written by the framework developer. Therefore, all the burden of learning a new language is placed on the expert framework developers, not the novice plugin developers. Additionally, a single framework developer writing specifications can now provide benefit for hundreds of plugin developers.

   While the framework developer receives little direct benefit for writing specifications, it might improve usage of the framework. It's also possible that third-party consultants, like those who are already answering questions on the forums, would be able to sell specification sets for an existing framework.

Composability of constraints. To further reduce the specification burden, Fusion allows framework developers to specify a single constraint at a time. The constraints are composable, and there is no need to specify the entire framework, or even an entire class, in order to get benefit from a single constraint. Additionally, as the analysis doesn't verify the framework itself, the constraints can be superficial and only on the API of the framework.

   I envision that Fusion could be used as a "firefighting" technique when writing a framework; instead of specifying the entire API up front, the developers can specify parts as needed based upon the struggles of plugin developers. For example, as it seems clear from the ASP.NET study that many plugin developer's problems were due to misuse of the `Page` lifecycle, this would be an ideal place to start writing specifications.

Localized errors. As seen, the analysis provides plugin developers with an error that directs them to the problem within their own code, rather than to where the problem is discovered at run time. The exact location is dependent on how the framework developer specifies it, but this makes sense as the framework developer is the expert for determining which expression was at fault.

Many options for different kinds of cost tradeoffs. Cost-effectiveness might vary based upon the kind of framework being used, the kind of plugin, or even the stage of development that the plugin is in. Fusion provides many different knobs to tune specifically to the needs of the system. For example, changing the amount of specifications, or even using automated specifications as

described in Chapter 7, can significantly increase the precision of the analysis. The precision can also be increased using a more precise points-to analysis. Of course, the three variants themselves also provide a tradeoff point, and Chapter 7 even suggests that while pragmatic may be good for less mature code, production code might benefit more from the complete analysis.

Chapter 5

Aliasing and Declarative Files

In the previous chapter, I showed how Fusion can specify and analyze collaboration constraints. While I mentioned that the Fusion analysis requires a points-to analysis, I elided all discussion of how this works. This chapter will more fully describe how Fusion uses the points-to analysis and the effect of the precision of the points-to analysis on the precision of the Fusion analysis.

This chapter will also introduce how Fusion handles declarative files. As we will see, the existence of declarative files negatively impacts the precision of the points-to analysis and the Fusion analysis. To regain this lost precision, I will introduce one last piece of the specification language, the `restrict-to` predicate. This predicate will allow relationships to specify important information about the aliasing of variables and will allow Fusion to be surprisingly precise in the presence of declarative files and imprecise points-to analyses.

5.1 Binding specification variables

To understand how the points-to analysis affects Fusion, it is first necessary to understand how Fusion will use the results. The points-to analysis will provide several potential aliasing configurations within the heap, and Fusion will use this information to evaluate constraints under all potential conditions.

To explore this further, I will formalize how this is done and then provide an intuitive understanding of how the analysis uses the points-to information. Recall that the points to lattice, $\mathcal{A}$ is defined as:

$$\begin{array}{rcl}
\mathcal{A} & ::= & < \Gamma_\ell; \mathcal{L} > \\
\Gamma_\ell & ::= & \{\ell : \tau\} \\
\mathcal{L} & ::= & \{\mathbf{x} \mapsto \{\ell\}\}
\end{array}$$

Also recall that a relationship R and the relationship lattice ρ are defined as:

$$\begin{array}{rcl}
\rho & ::= & \{R \mapsto t\} \\
R & ::= & \mathsf{rel}(\bar{\ell}) \\
t & ::= & \mathsf{True} \mid \mathsf{False} \mid \mathsf{Unknown}
\end{array}$$

While all of these definitions use ℓ, specification predicates are written not on a runtime label, but on a specification variable, written as y. These are different from the source variables, written as x.

$$P \quad ::= \quad P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid \neg A \mid A \mid \mathsf{True} \mid \mathsf{False}$$
$$A \quad ::= \quad \mathsf{rel}(\bar{\mathsf{y}})$$

Therefore, to evaluate the truth of a specification predicate, we rely on a substitution σ that replaces each y with a ℓ.

$$\sigma \quad ::= \quad \{\mathsf{y} \mapsto \ell\}$$

This will allow the analysis to evaluate a predicate using the judgment below; the rules for this judgment are in three-value logic and described further in Figures B.19-B.22 of Appendix B. In this judgment, $P[\sigma]$ represents the specification predicate with each y substituted by the ℓ mapped in σ.

$$\rho \vdash P[\sigma]\; t$$

To evaluate the judgment above, Fusion will need to produce all possible substitutions for each constraint. Specifically, it will use the points-to lattice to generate two sets of σ.

The first set represents the substitutions that are possible *without* considering the requires predicate. This set will be created using the function findLabels, defined in Figure 5.1. This function takes:

1. the points-to lattice $\mathcal{A}$,

2. β, which is a mapping of specification variables y to source variables x for every specification variable in the operator of the constraint, and

3. Γ_{y}, which is a typing context for a set of specification variables, in this case, all specification variables *except those used exclusively by* P_{req}.

The details for how β and Γ_{y} are created are beyond the scope of this discussion and can be found in Appendix B, but they are created in a straightforward and expected way. The purpose of this function is to find all substitutions such that every y in Γ_{y} has a ℓ with a substitutable type and that any y in β only uses the labels pointed to by the corresponding source variable x.

The second set represents the substitutions *including* the requires predicate. This will be created using the function allValidSubs, as defined in Figure 5.1. This function will also take the points-to lattice and a specification typing context. However, it also takes an existing substitution context σ that it should extend. The function will find all substitutions that will extend σ so that the resulting substitutions have the same domain as Γ_{y} and so that they all satisfy the types defined in Γ_{y}.

With these two sets, Fusion can now check a given constraint under a particular points-to lattice and relationship lattice. As shown previously in Table 4.1, the three variants check constraints differently with respect to when the predicates can be true. As we will see now, they are also different with respect to how they select a substitution to check. Intuitively, σ represents a possible heap configuration at run time. Therefore, we will expect that the sound variant checks all possible heaps, and the complete variant will only check a known subset.

$\boxed{\mathsf{findLabels}(< \Gamma_\ell; \mathcal{L} >; \beta; \Gamma_y) = \Sigma}$

$$\Sigma = \{\sigma' \mid \sigma = \{y \mapsto \ell \mid y \in \mathrm{dom}(\beta) \wedge \ell \in \mathcal{L}(\beta(y)) \wedge \exists \tau' . \tau' <: \Gamma_\ell(\ell) \wedge \tau' <: \Gamma_y(y)\} \wedge$$

$$\sigma' \in \mathsf{allValidSubs}(< \Gamma_\ell; \mathcal{L} >; \sigma; \Gamma_y)\}$$

$\boxed{\mathsf{allValidSubs}(< \Gamma_\ell; \mathcal{L} >; \sigma; \Gamma_y) = \Sigma}$

$$\Sigma = \{\sigma' \mid \sigma' \supseteq \sigma \wedge \mathrm{dom}(\sigma') = \mathrm{dom}(\Gamma_y) \wedge \forall y \mapsto \ell \in \sigma' . \exists \tau' . \tau' <: \Gamma_\ell(\ell) \wedge \tau' <: \Gamma_y(y)\}$$

Figure 5.1: Functions for generating the substitutions. $\mathsf{findLabels}$ uses β to create substitutions for all specification variables that are bound to a source variable, then uses $\mathsf{allValidSubs}$ to generate substitutions for the remaining, unbound specifications variables in Γ_y.

For the *sound* variant, we will check an instruction for a single constraint, $\mathsf{op} : P_{trg} \Rightarrow P_{req} \Downarrow \bar{A};$. Let β be a binding between the variables of op and the instruction, and let ρ and $\mathcal{A}$ be the entry lattices such that ρ is consistent with $\mathcal{A}$. Also let Γ_y^{noreq} be the free variables in op, P_{trg}, and $\bar{A}$, and let Γ_y^{req} be all the free variables in the constraint, including those in P_{req}. The sound variant will then check the constraint by ensuring that the following predicate is true:

$$\forall \sigma \in \mathsf{findLabels}(\mathcal{A}; \beta; \Gamma_{y_{noreq}}) . \rho \vdash P_{trg}[\sigma]t \wedge t \neq \mathsf{False} \implies$$
$$\forall \sigma' \in \mathsf{allValidSubs}(\mathcal{A}; \sigma; \Gamma_{y_{req}}) . \rho \vdash P_{req}[\sigma']\mathsf{True}$$

The sound variant must ensure that there are no false negatives (with respect to the given lattice $\mathcal{A}$). Therefore, as any of the possible substitutions can occur at run time, it must check all of them and uses two universal quantifiers.

The reader might notice that the second quantifier above is redundant; we could have instead written:

$$\forall \sigma \in \mathsf{findLabels}(\mathcal{A}; \beta; \Gamma_{y_)} . \rho \vdash P_{trg}[\sigma]t \wedge t \neq \mathsf{False} \implies \rho \vdash P_{req}[\sigma]\mathsf{True}$$

The two quantifiers are necessary because this is another key point where the variants are different. As shown below, the complete variant uses an existential for the second quantifier.

$$\forall \sigma \in \mathsf{findLabels}(\mathcal{A}; \beta; \Gamma_{y_{noreq}}) . \rho \vdash P_{trg}[\sigma]\mathsf{True} \implies$$
$$\exists \sigma' \in \mathsf{allValidSubs}(\mathcal{A}; \sigma; \Gamma_{y_{req}}) . \rho \vdash P_{req}[\sigma']t \wedge t \neq \mathsf{False}$$

The complete variant must ensure that there are no false positives (with respect to the given lattice $\mathcal{A}$). As the analysis is not sure under what conditions the requires predicate might be true, the constraint will pass as long as there exists *some* possibility of the constraint passing at run time. Why isn't the first quantifier an existential as well then? The complete variant is not complete with respect to the entire program, rather, it is complete *with respect to a given aliasing configuration* (as preselected by the function $\mathsf{findLabels}$). Since this function is binding the specification variables that matched to a source variable, it is starting with only those substitutions that the points-to analyses deems to be possible. A more precise points-to analysis would increase this precision.

The pragmatic analysis follows the complete variant in this case, though of course it uses its own rules for checking P_{req} as described earlier in Table 4.1. In practice, the second set produces a significant number of substitutions since the variables are bound to any known object label with the right type. The likelyhood of all of these passing is low, and in practice, this is the source of many false positives in the sound variant. Therefore, the pragmatic analysis will work as shown below:

$$\forall \sigma \in \mathsf{findLabels}(\mathcal{A}; \beta; \Gamma_{y_{noreq}}) \; . \; \rho \vdash P_{trg}[\sigma]\mathsf{True} \implies$$
$$\exists \sigma' \in \mathsf{allValidSubs}(\mathcal{A}; \sigma; \Gamma_{y_{req}}) \; . \; \rho \vdash P_{req}[\sigma']\mathsf{True}$$

The above checking is done for each constraint in the system, and any failures for a constraint to meet the given predicate causes an error to the user for that combination of constraint and source instruction. At present, the particular substitution that caused the failure is not reported, but that could be easily added with an appropriate reporting capability to explain the substitution that causes the error to the user.

5.2 Creating effects

Once the flow function has created substitutions and checked the constraint, it needs to use those substitutions to create any effects. Unlike the previous section, we will now only need to use the first set of substitutions created by $\mathsf{findLabels}$, as it contains all the specification variables used by the effects $\bar{A}$. Additionally, all the variants work the same when producing the effects. In this section, I'll describe how effects are created by starting with a single σ for a single constraint and then working upwards until we change the original lattice ρ to create a new lattice ρ'. A more formal description of this is available in Appendix B.

The first step is to create the effects for a single σ. In all variants, if $\rho \vdash P_{trg}[\sigma]\mathsf{True}$, then the effects $\bar{A}[\sigma]$ will be created. However, if $\rho \vdash P_{trg}[\sigma]\mathsf{Unknown}$, then the effects $\bar{A}[\sigma]$ will still be created but marked as coming from an $\mathsf{Unknown}$ with a $*$. For example, if we have a constraint with effect $\mathsf{Selected(item)}$, then when the trigger is True with substitution σ, the analysis will produce $\mathsf{Selected(item)}[\sigma] \mapsto \mathsf{True}$, but if the trigger was $\mathsf{Unknown}$, it will produce $\mathsf{Selected(item)}[\sigma] \mapsto \mathsf{True}*$. This marker will be used later when determining how to handle $\mathsf{Unknown}$ predicates without losing further precision.

When each σ from $\mathsf{findLabels}$ has produced a set of effects, they must be merged together. Any conflicts, such as True and False, are resolved to $\mathsf{Unknown}$. Additionally, starred effects propagate themselves, that is, merging True and $\mathsf{True}*$ produces $\mathsf{True}*$. The rationale behind this is that if one substitution produces a True, we cannot be sure that this substitution will be used at run time. The other substitution that produces $\mathsf{True}*$ may be used instead. This other substitution also has an $\mathsf{Unknown}$ trigger, which may be False at run time. Therefore, it is important to preserve this possibility so as not to change the effect to True when it may not actually be the case.

Once each constraint has a set of effects, they have to be merged together as well. At this level, the constraints are merged slightly differently than above. Unlike the substitutions, where only one is possible, we know that all constraints exist at all times. Therefore, they are treated as independent events that may change the effects. This means that they can still conflict and

Table 5.1: Sample of rules for the flow function of the two points-to analyses. Assumes a variable typing environment Γ_x and the subtyping relation $<:$. The differences are shaded.

instr	$f_{may-like}(<\Gamma_\ell; \mathcal{L}>, instr)$	$f_{must-like}(<\Gamma_\ell; \mathcal{L}>, instr)$
$x_1 = x_2$	$<\Gamma_\ell;$	$<\Gamma_\ell;$
	$\mathcal{L}[x_2 \mapsto \mathcal{L}(x_1)] >$	$\mathcal{L}[x_2 \mapsto \mathcal{L}(x_1)] >$
$x = new\ C(\bar{x})$	$<\Gamma_\ell, \ell_{fresh} : C;$	$<\Gamma_\ell, \ell_{fresh} : C;$
	$\mathcal{L}[x \mapsto \{\ell_{fresh}\}] >$	$\mathcal{L}[x \mapsto \{\ell_{fresh}\}] >$
$x_1 = x_2.method(\bar{x}))$	$<\Gamma_\ell, \ell_{fresh} : \Gamma_x(x_1);$	$<\Gamma_\ell, \ell_{fresh} : \Gamma_x(x_1);$
	$\mathcal{L}[x_1 \mapsto \{\ell \mid \Gamma_\ell(\ell) <: \Gamma_x(x_1)\} \cup \{\ell_{fresh}\}] >$	$\mathcal{L}[x_1 \mapsto \{\ell_{fresh}\}] >$
$x_1 = x_2.field$	$<\Gamma_\ell, \ell_{fresh} : \Gamma_x(x_1);$	$<\Gamma_\ell, \ell_{fresh} : \Gamma_x(x_1);$
	$\mathcal{L}[x_1 \mapsto \{\ell \mid \Gamma_\ell(\ell) <: \Gamma_x(x_1)\} \cup \{\ell_{fresh}\}] >$	$\mathcal{L}[x_1 \mapsto \{\ell_{fresh}\}] >$

resolve to Unknown, however, merging True from one constraint and True$*$ from another will now produce True.

Finally, the effects must be applied to the original ρ using a weak update. Any non-starred effects are applied directly. Starred effects will cause the relationship to change to Unknown *unless* the original relationship in ρ has the same state as the base of the starred effect. This prevents an unnecessary loss of precision in cases where the effect is actually maintaining the status quo.

5.3   Points-to analysis

The previous two sections described how the analysis uses the points-to lattice to generate a set of substitutions σ to check the requires predicate and generate relationship effects. In this section, we explore how the results of the points-to lattice can directly affect the precision of the Fusion analysis in practice. We will explore this using a single variant of the analysis (pragmatic) with two different points-to analyses. The first points-to analysis is akin to a may-alias analysis, while the second is similar to a must-alias analysis. Table 5.1 shows a selection of transfer functions for the analyses to highlight their differences. The primary difference is that the may-like analysis adds in all known labels ℓ that satisfy the type τ of the source variable x, whereas the must-like analysis assumes unique references unless it explicitly discovers otherwise.

In the results from Table 4.2, I used the may-like analysis for the sound variant (as it is sound itself) and the must-like analysis for the complete variant. In this table, the pragmatic variant also used the must-like analysis. Table 5.2 shows only the results for pragmatic, but with both points-to analyses. Notice that both catch all the errors, but the may-like analyses causes false positives in both of the correct examples (though still not as many as the sound variant).

Let's explore the correct selection example to see what happened. We'll use the pragmatic variant with the may-like points-to analysis. Listing 5.1 shows both the lattices in comments between each line.

Table 5.2: Results from running the pragmatic variant with different points-to analyses on the examples from Vignette 3.1

Listing reference	Line number of fault	Pragmatic results (may-like)	Pragmatic results (must-like)
3.1: Naïve selection	7	7	7
3.2: Correct selection	-	9	-
3.3: Forgotten deselection	14	14	14
3.4: Nothing selected	14	14	14
3.5: Two lists, incorrect	13	13	13
(Not given): Two lists, correct	-	13	-
3.6: Swapped selection	$7, 9$	$7, 9$	$7, 9$

Listing 5.1: Correct usage of a `DropDownList` run with the may-like points-to analysis and the pragmatic variant of Fusion, with $\mathcal{A}$ and ρ in comments.

```
1   DropDownList list;
2
3   private void Page_Load(object sender, EventArgs e)
4   {
5     ListItem newSel, oldSel;
6     ListItemCollection coll;
7   //<l₁:DropDownList ; list−>{l₁}>
8   //−
9     oldSel = list.getSelectedItem();
10  //<l₁:DropDownList, l₂:ListItem ; list↦{l₁}, oldSel↦{l₂}>
11  //Selected(l₂), Child(l₂, l₁)
12    oldSel.setSelected(false);
13  //<l₁:DropDownList, l₂:ListItem ; list↦{l₁}, oldSel↦{l₂}>
14  //!Selected(l₂), Child(l₂, l₁), !CorrectlySelected(l₁)
15    coll = list.getItems();
16  //<l₁:DropDownList, l₂:ListItem, l₃: ListItemCollection ; list↦{l₁}, oldSel↦{l₂}, coll↦{l₃}>
17  //!Selected(l₂), Child(l₂, l₁), !CorrectlySelected(l₁), Items(l₃, l₁)
18    newSel = coll.findByValue("foo");
19  //<l₁:DropDownList, l₂:ListItem, l₃: ListItemCollection, l₄: ListItem ;
20  //list↦{l₁}, oldSel↦{l₂}, coll↦{l₃}, newSel↦{l₂,l₄}>
21  //!Selected(l₂), Child(l₂, l₁), !CorrectlySelected(l₁), Items(l₃, l₁)
22    newSel.setSelected(true);
23  //<l₁:DropDownList, l₂:ListItem, l₃: ListItemCollection, l₄: ListItem ;
24  //list↦{l₁}, oldSel↦{l₂}, coll↦{l₃}, newSel↦{l₂,l₄}>
25  //Child(l₂, l₁), Items(l₃, l₁)
26  }
```

Everything works as expected until we get to line 18. Notice that at this instruction, the points-to analysis will need to decide what `newSel` can point to. Since it is not sure whether or not it aliases `oldSel`, it points to both ℓ_2 and ℓ_4. Therefore, Fusion will have to run the analysis with both of these possibilities, and it will create effects for two possible substitutions.

$$\sigma_1 = \{newSel \mapsto \ell_2, ctrl \mapsto \ell_1, coll \mapsto \ell_3\} \text{ produces } \mathsf{Child}(\ell_2, \ell_1) \mapsto \mathsf{True}$$

$$\sigma_2 = \{newSel \mapsto \ell_4, ctrl \mapsto \ell_1, coll \mapsto \ell_3\} \text{ produces } \mathsf{Child}(\ell_4, \ell_1) \mapsto \mathsf{True}$$

When these substitutions are merged together, both relationships will go to True*.

Therefore, as only $\mathsf{Child}(\ell_2, \ell_1)$ is True in ρ, only this effect will remain, and $\mathsf{Child}(\ell_4, \ell_1)$ will be lost.

At first, this lack of precision will cause no problems. Line 21 will still verify correctly for the second constraint in Listing 4.4 with both substitutions as σ_1 will cause both the trigger and required predicate to evaluate to True, and the σ_2 will cause an Unknown trigger so the requires predicate will not be checked However, it also means that both of these substitutions will produce effects on the relationship $\mathsf{CorrectlySelected}(\ell_1)$, and the second substitution will be setting it to True* because its trigger was unknown. Both will again merge to True*, but as the relationship exists in ρ as False, it will be changed to Unknown, not True. When the analysis reaches the end of the method, it will attempt to verify the final constraint in Listing 4.4. As the trigger is True but the requires predicate is now Unknown, the pragmatic variant using a may-like points-to analysis will give an error.

The type of problem described above will occur in any situation where the code has two variables of the same type, which is why the problem also appears in the correct example with two `DropDownLists`. The must-like analysis simply avoids this by assuming the uniqueness of pointers unless otherwise specified, which reduces the number of substitutions used by the Fusion analysis.

There are two possible solutions to this problem, which are beyond the scope of this thesis. The first solution is to improve the points-to analysis, either through deeper analysis techniques or through specifications. Much research has been done in these areas [21, 22, 68, 76, 95], so using a more sophisticated analysis would certainly be feasible. The second solution is to keep separate lattices for each potential heap configuration so that they do not ever merge and loose precision. Doing so would require more implementation effort and may cause an exponential blowup in large methods, thus limiting the scalability of the analysis.

The important issue to take away from this section is that *every additional label in the points-to lattice can cause later imprecision*. This issue will become more relevant later in this chapter.

5.4 Getting relationships from declarative artifacts

We'll now leave behind points-to analyses, aliases, and labels for a section to discuss the use of declarative files in Fusion. Don't despair though, we will be back to the complexities of aliasing shortly.

Chapter 2 introduced the concept of *declarative artifacts* and how software frameworks use these declarative artifacts to increase their flexibility. While these artifacts are increasingly com-

Listing 5.2: ASPX with a `LoginView`

```
1    <asp:LoginView ID="LoginScreen" runat="server">
2      <AnonymousTemplate>
3         You can only set up your account
4         when you are logged in.
5      </AnonymousTemplate>
6      <LoggedInTemplate>
7         <h4>Location</h4>
8         <asp:DropDownList ID="LocationList"
9            runat="server"/>
10        <asp:Button ID="ContinueButton"
11           runat="server" Text="Continue"/>
12     </LoggedInTemplate>
13   </asp:LoginView>
```

Listing 5.3: Incorrect way of retrieving controls in a `LoginView`

```
1    LoginView LoginScreen;
2
3    private void Page_Load(object sender, EventArgs e)
4    {
5       if (!isPostback()) {
6          DropDownList list = (DropDownList)
7           LoginScreen.FindControl("LocationList");
8          list.DataSource = ...;
9          list.DataBind();
10      }
11   }
```

mon, no known general purpose verification technique can handle these files alongside the program code. Of course, several types of declarative artifacts provide basic checking (such as schemas for XML), and many frameworks provide custom verification, built into the IDE, that provide basic checking for their own artifacts (like the ASP.NET, Eclipse, and Spring frameworks). There are also many research proposals to increase the amount of verification for a given artifact, for example, adding typechecking to XML. Finally, there are two research proposals that verify declarative files with code for a specific framework [3, 7], but there is nothing for general purpose checking. As we will see, it is absolutely necessary to verify declarative files with their associated program code rather than verifying them separately.

Consider the example with the `LoginView`, as described in Vignette 2.2. By themselves, both the code in Listing 5.3 and the declarative ASPX file in Listing 5.2 look correct, and traditional verifiers would check this appropriately. However, when viewed together, there is clearly a problem because the `DropDownList` is inside the `LoginView`'s `LoggedInTemplate`.

As presented in Chapter 4, Fusion would also not be able to properly verify the incorrect and correct versions of this program. Specifying the API is straightforward and is shown in Listing 5.4. The constraint on `LoginView.findControl(String)` says that if the requested control is in

Listing 5.4: Specifications for correct usage of `LoginView.findControl(String)`

```
1   public class Control {
2     public Control findControl(String name) {...}
3     ...
4   }
5
6   @Constraint(
7     op="LoginView.findControl(String name) : Control",
8     trigger = "Name(name, result) AND LoggedInControl(result, target)",
9     requires = "SubControl(target, page) AND PageRequest(request, page) AND Authenticated(request)",
10    effects = {}
11  )
12  @Constraint(
13    op="LoginView.findControl(String name) : Control",
14    trigger = "Name(name, result) AND AnonymousControl(result, target)",
15    requires = "SubControl(target, page) AND PageRequest(request, page) AND !Authenticated(request)",
16    effects = {}
17  )
18  public class LoginView extends Control {
19    ...
20  }
21
22  public class Page extends Control {
23   @PageRequest({result, target}, ADD)
24    public Request getRequest() {...}
25      ...
26  }
27
28  public class Request {
29   @Authenticated({this}, TEST, result)
30    public boolean isAuthenticated() {...}
31      ...
32  }
```

the `LoggedInTemplate`, we must know that a user is logged in. However, this requires us to have a LoggedInControl relationship with the appropriate parameters, and this relationship cannot be generated with the program code shown, even in the correct program in Listing 2.4.

While the LoggedInControl relationship does not exist in the program code, it *does* exist in the ASPX file in Listing 5.2. In this file, the requested `DropDownList` is clearly inside the `LoggedIn-Template`. Therefore, we must somehow extract this relationship from the ASPX.

To get these relationships, Fusion supports using XQuery to query XML-based artifacts for relationships. These relationships are then used as the starting lattice ρ before analyzing any program code. While Fusion currently only supports XML-based files, a similar extraction mechanism could be used for other file types as well.

The XQuery for retrieving the relationships SubControl, LoggedInControl, and Anonymous-Control are shown in Listing 5.5. This listing first defines several locals used to get the names and types of the elements, then it declares four queries that retrieve the relationships. While these

Listing 5.5: XQuery to retrieve the relationships SubControl, LoggedInControl, and Anonymous-Control

```
1  declare namespace asp="aspx";
2  declare namespace fusion="http://code.google.com/p/fusion";
3  declare variable $doc as xs:string external;
4
5  declare function local:type($element as node()) as xs:string {
6      if (local-name($element) = "Page" and namespace-uri($element) = "aspx")
7          then $element/@codebehind
8        else concat("edu.cmu.cs.fusion.test.aspnet.api.",local-name($element))
9  };
10
11 let $page := doc($doc)/asp:Page/.
12 for $control in $page/asp:*
13 where fusion:isSubtype(local:type($control), "edu.cmu.cs.fusion.test.aspnet.api.Control")
14 return <Relationship name="SubControl" effect="ADD">
15      <Object name ="{data($control/@ID)}" type="{local:type($control)}"/>
16      <Object name="{data($page/@ID)}" type="{local:type($page)}"/>
17    </Relationship>
18
19 let $page := doc($doc)/asp:Page/.
20 for $control in $page/asp:*
21 for $subControl in $control/asp:*
22 where fusion:isSubtype(local:type($control), "edu.cmu.cs.fusion.test.aspnet.api.Control") and
23   fusion:isSubtype(local:type($subControl), "edu.cmu.cs.fusion.test.aspnet.api.Control")
24 return <Relationship name="SubControl" effect="ADD">
25      <Object name ="{data($subControl/@ID)}" type="{local:type($subControl)}"/>
26      <Object name ="{data($control/@ID)}" type="{local:type($control)}"/>
27    </Relationship>
28
29 let $page := doc($doc)/asp:Page/.
30 for $control in $page/asp:LoginView
31 for $subControl in $control/AnonymousTemplate/asp:*
32 where fusion:isSubtype(local:type($subControl), "edu.cmu.cs.fusion.test.aspnet.api.Control")
33 return <Relationship name="AnonymousControl" effect="ADD">
34      <Object name ="{data($subControl/@ID)}" type="{local:type($subControl)}"/>
35      <Object name ="{data($control/@ID)}" type="{local:type($control)}"/>
36    </Relationship>
37
38 let $page := doc($doc)/asp:Page/.
39 for $control in $page/asp:LoginView
40 for $subControl in $control/LoggedInTemplate/asp:*
41 where fusion:isSubtype(local:type($subControl), "edu.cmu.cs.fusion.test.aspnet.api.Control")
42 return <Relationship name="LoggedInControl" effect="ADD">
43      <Object name ="{data($subControl/@ID)}" type="{local:type($subControl)}"/>
44      <Object name ="{data($control/@ID)}" type="{local:type($control)}"/>
45    </Relationship>
46
47 let $page := doc($doc)/asp:Page/.
48 return <ThisObject name="{data($page/@ID)}" type="{local:type($page)}"/>
```

are unwieldy looking specifications, they would be used for all plugins of a given framework, so the specification cost is amortized.[1] Fusion also supports the ability to bind the `this` variable to an object in the declarative artifact; this XQuery is shown at the bottom of Listing 5.5 A similar mechanism could also be used to bind fields.

When the XQuery from Listing 5.5 is run on the ASPX from Listing 5.2, Fusion will get a starting lattice as shown:

$$\mathrm{SubControl}(\mathtt{LoginScreen}, \mathtt{MyPage}) \mapsto \mathsf{True}$$

$$\mathrm{LoggedInControl}(\mathtt{LocationList}, \mathtt{LoginScreen}) \mapsto \mathsf{True}$$

$$\mathrm{LoggedInControl}(\mathtt{ContinueButton}, \mathtt{LoginScreen}) \mapsto \mathsf{True}$$

This lattice will then allow Fusion to have the relationships necessary to verify the correct code and find the error in the broken code.

5.5 Impact of more labels

When the XQuery runs, it influences the Fusion analysis by creating a starting relationship lattice ρ. These relationships must refer to labels in the points-to lattice; therefore the XQuery will also affect the starting points-to lattice $\mathcal{A}$. As we might guess based upon earlier sections, these additional labels are going to impact the precision of the points-to analysis.

Let's start by considering how the may-like points-to analysis runs on a very simple code snippet. Listing 5.6 shows this code snippet with $\mathcal{A}$ in the comments. As expected, the may-like points to analysis shows two cases: either `barList` points to the same object as `fooList` or it points to a different object. This is a small loss in precision, but it is still manageable.

Now consider what happens when we associate the code with the ASPX in Listing 5.7. This will create a starting $\mathcal{A}$ that contains two labels, representing the two `DropDownLists` in the ASPX. Listing 5.8 shows what happens to the points-to lattice when run on the code snippet now. While `fooList` still only points to a single fresh label (since it was created by constructor), the `barList` could now point to any one of four possible objects: the same object as `fooList`, one of the two lists in the ASPX, or some yet-unseen list. More knowledge from the ASPX file has made the analysis significantly *less* precise rather than more precise.

We might think we could solve this problem as we did earlier by switching to the must-like analysis. However, recall that this analysis assumes uniqueness for all variables, so it will only give `barList` the option of pointing to a fresh label, as seen in Listing 5.9. Clearly, this is not the programmer's intent either.

What we really want is to tell the points-to analysis that the *only* valid label is the bar label, since that's the object we requested with the call to `findControl`. However, there is no way for the points-to analysis to know this expected semantics. Even if we were to use a more sophisticated

---

[1]Part of the ugliness is due to the ugliness of XML itself and its inappropriateness for being used for this purpose in the first place. C'est la vie.

Listing 5.6: A simple code snippet, with the may-like points-to analysis.

```
1   //<−;−>
2   DropDownList fooList = new DropDownList();
3   //<l₁:DropDownList ; fooList↦{l₁}>
4   DropDownList barList = (DropDownList) findControl("bar");
5   //<l₁:DropDownList, l₂:DropDownList ; fooList↦{l₁}, barList↦{l₁,l₂}>
```

Listing 5.7: An ASPX file associated with code snippet from 5.6.

```
1   <asp:Content ID="Content1" ContentPlaceHolderID="PageContent">
2       <asp:DropDownList ID="bar">
3       <asp:DropDownList ID="baz"/>
4   </asp:Content>
```

Listing 5.8: Our code snippet again, now associated with the ASPX from Listing 5.7.

```
1   //<bar:DropDownList, baz:DropDownList ; − >
2   DropDownList fooList = new DropDownList();
3   //<bar:DropDownList, baz:DropDownList, l₁:DropDownList ; fooList↦{l₁}>
4   DropDownList barList = (DropDownList) findControl("bar");
5   //<bar:DropDownList, baz:DropDownList, l₁:DropDownList, l₂:DropDownList ;
6   // fooList↦{l₁}, barList↦{l₁, l₂, bar, baz}>
```

Listing 5.9: Using the must-like analysis doesn't do what we want either.

```
1   //<bar:DropDownList, baz:DropDownList ; − >
2   DropDownList fooList = new DropDownList();
3   //<bar:DropDownList, baz:DropDownList, l₁:DropDownList ; fooList↦{l₁}>
4   DropDownList barList = (DropDownList) findControl("bar");
5   //<bar:DropDownList, baz:DropDownList, l₁:DropDownList, l₂:DropDownList ; fooList↦{l₁}, barList↦{l₂}>
```

Table 5.3: Results from running each variant on the examples from Vignette 2.2.

Listing reference	Line number of fault	Sound results	Pragmatic results, may-like	Pragmatic results, must-like	Complete results
2.3: Incorrect usage	6	6, 6	6	-	-
2.4: Correct usage	-	7, 7	-	-	-

points-to analysis, it would not ensure that we get the right object back from `findControl`; the most we could expect is to be able to specify is that `fooList` and `barList` do not alias.

5.6 The restrict predicate

The problem in the prior section was that the points-to analysis has no way to select out a particular object from a group of objects. Fusion solves this by using relationships to specify which labels make sense. For example, what we really want to say about is that the returned object from `Control.findControls(String name)` satisfies the predicate:

$$\text{Name}(\texttt{name}, \texttt{result}) \wedge \text{SubControl}(\texttt{result}, \texttt{target})$$

That is, the returned object is a sub-control of the object we called `findControl` on and it has the name we are searching for.

To support this, Fusion constraints will contain one more predicate, the restrict-to predicate. An example of this predicate can be seen in the `Control` API constraints in Listing 5.10. The semantics of this predicate is when the trigger predicate is True, the analysis will restrict the potential substitutions to only those that pass the restrict-to predicate. The sound and complete variants will only restrict a False predicate, while the pragmatic variant will restrict either False or Unknown. The formal semantics of this predicate can be found in Appendix B. In practice, this predicate is frequently Unknown, but the sound and complete variants are not sound or complete unless they accept an Unknown restrict-to predicate.

With this in place, the analysis can now finally verify programs that use declarative artifacts. Listing 5.11 shows the snippet run with the restrict-to predicates described above; notice that now `barList` only points to the single `DropDownList` with the name bar, as we expected. This also allows us to finally verify the examples from Vignette 2.2; Table 5.3 provides the results for running the analysis with three variants, including the pragmatic variant with both versions. As the restrict-to predicate makes the may-like analysis a practical option, we will use the may-like analysis with the pragmatic variant for the remainder of the thesis.

As seen, a few points of variation in these analyses makes a large difference in their results. Table 5.4 lists all the differences between the three variants of the Fusion analysis.

Listing 5.10: Constraining `LoginView.findLabels(String)` with a restrict-to predicate.

```
@Constraint(
    op="Control.findControl(String name) : Control",
    trigger = "True",
    restrict−to = "Name(name, result) AND SubControl(result, target)",
    requires = "True",
    effects = {}
)
public class Control {
    public Control findControl(String name) {...}
    ...
}

@Constraint(
    op="LoginView.findControl(String name) : Control",
    trigger = "Name(name, result) AND LoggedInControl(result, target)",
    requires = "SubControl(target, page) AND PageRequest(request, page) AND Authenticated(request)",
    effects = {}
)
@Constraint(
    op="LoginView.findControl(String name) : Control",
    trigger = "Name(name, result) AND AnonymousControl(result, target)",
    requires = "SubControl(target, page) AND PageRequest(request, page) AND !Authenticated(request)",
    effects = {}
)
public class LoginView extends Control {
    ...
}
```

Listing 5.11: Using the restrict-to predicate as seen in Listing 5.10 to get the aliasing that we want with the may-like points-to analysis

```
//<bar:DropDownList, baz:DropDownList ; − >
DropDownList fooList = new DropDownList();
//<bar:DropDownList, baz:DropDownList, l₁:DropDownList ; fooList↦{l₁}>
DropDownList barList = (DropDownList) findControl("bar");
//<bar:DropDownList, baz:DropDownList, l₁:DropDownList ; fooList↦{l₁}, barList↦{bar}>
```

Table 5.4: All differences between sound, complete, and pragmatic variants.

Variant	Trigger predicate checks when	Requires quantifies σ with	Requires predicate passes when	Restrict-to allows σ when
Sound	True/Unknown	$\forall$	True	True/Unknown
Complete	True	$\exists$	True/Unknown	True/Unknown
Pragmatic	True	$\exists$	True	True

Chapter 6

Case Study: Spring Framework

To validate that Fusion is a general tool for specifying collaboration constraints, I studied how Fusion can be used to specify the Spring framework, a framework with a surprisingly different design from ASP.NET. In this chapter, I'll present the methodology of this study and some quantitative results that compare the variants of Fusion. I'll also present four collaboration constraints in Spring that I specified with Fusion and use them to highlight several interesting tradeoffs that occur when using Fusion.

Based on this study, there is good reason to believe that relationship-based specifications can be used to specify collaboration constraints within software frameworks that use a wide variety of mechanisms to interact with plugins. Overall, I had to make very few changes to Fusion to be able to specify the collaboration constraints described in this chapter. There are several features that would allow for more collaboration constraints to be specified, but all of these are engineering efforts that would not require any additional research contributions.

6.1 Why Spring

When selecting a framework for this case study, I considered several criteria. First, the framework had to be written in Java and XML, as Fusion currently only supports those languages. Second, I chose not to use a framework that I was already familiar with in order to prevent unintentional bias from seeing similar collaboration constraints before starting the evaluation. Third, I wanted to use a framework which was large, complex, and uses several mechanisms to interact with plugin code, not just traditional OO mechanisms. Finally, I wanted a framework with a large enough following to have an active community forum from which I could draw examples. The Spring Framework fit all of these criteria.

The primary downside to using Spring as a case study is that it is a competitor to ASP.NET. Both frameworks are web application frameworks, meant to help developers build large industrial web applications. In theory, this shared domain might mean similar architecture and design of the framework, which might result in similar collaboration constraints. However, I found that the two frameworks are quite different from each other at nearly every level of abstraction. While

both frameworks use the model-view-controller pattern to represent a request for a web page and responding with the HTML for this request, the similarities end there. The frameworks have completely different structures to their APIs, different mechanisms for connecting several pages into a web application, and different reuse capabilities for common tasks. The reason for all these differences is because the two framework have nearly opposite business drivers. This completely changes how the frameworks are architected, and the differences trickle down into even low-level design decisions.

In ASP.NET, the primary business driver is simple: keep the client using as many Microsoft technologies as possible. In fact, ASP.NET will generally only work with other Microsoft products: the plugin developer must deploy their application using Microsoft's web server running on a Microsoft operating system, and likely using a Microsoft database. Even the development is controlled by Microsoft: the languages, IDE, and build systems are all required Microsoft products, and many shops will use Microsoft source repositories and project management software as well.

All this control over every aspect of development and deployment means that Microsoft can make many assumptions about the environment in order to simplify the design of ASP.NET. For instance, there's no need for generic interfaces to many components when there is only one option. The framework can also take advantage of the IDE control and use tools to auto-generate common code and provide WYSIWYG editors for creating the UI of a page. This all leads to smaller, cleaner APIs. Of course, the plugin developers must be prepared to fully buy-in to Microsoft and might not be able to interact easily with legacy systems, but Microsoft hopes that such systems will be converted and further lock the application to Microsoft.

Spring takes a very different approach to attracting customers. Instead of locking in clients, Spring aims to support a wide variety of legacy systems and be as interoperable as possible. VMWare, the owners of Spring, boast that "Spring provides a *range of capabilities* for creating enterprise Java, rich web, and enterprise integration applications that can be consumed in a *lightweight, a-la-carte manner*." [101] Each component of Spring can be used independently or can be replaced by a third-party component, and it is assumed that developers will be integrating with an existing third-party web application framework. The book "Pro Spring", written by a member of the Spring team, devotes a chapter on how to integrate Spring with Struts, the next most popular Java web application framework [51]. Both the official Spring reference manual, and a second popular Spring book, go further by describing how to integrate Spring with Struts, WebWork, Tapestry, and Java Server Faces [61, 113].

Even the language used to create the views is modular in Spring. While views in ASP.NET are always written in ASPX, Spring views can be created from many different technologies. While JSP is popular, both of the books above dedicate a chapter to describing other technologies with Spring, such as Velocity, Tiles, RSS, and even how to integrate with a custom technology.

While Spring provides a great amount of flexibility, the cost to the design is high. Each point of variability must be behind an API, and the API must be as generic as possible. In order to promote reuse then, the class hierarchies are necessarily deep so that the most generic API is at the top of the hierarchy and the most specific APIs are at the leaves. As an example, consider the controller hierarchy in Figure 6.1. The top most interface has a single method, which is certainly more simple than the `Page` API in ASP.NET. This interface provides no code reuse capability and
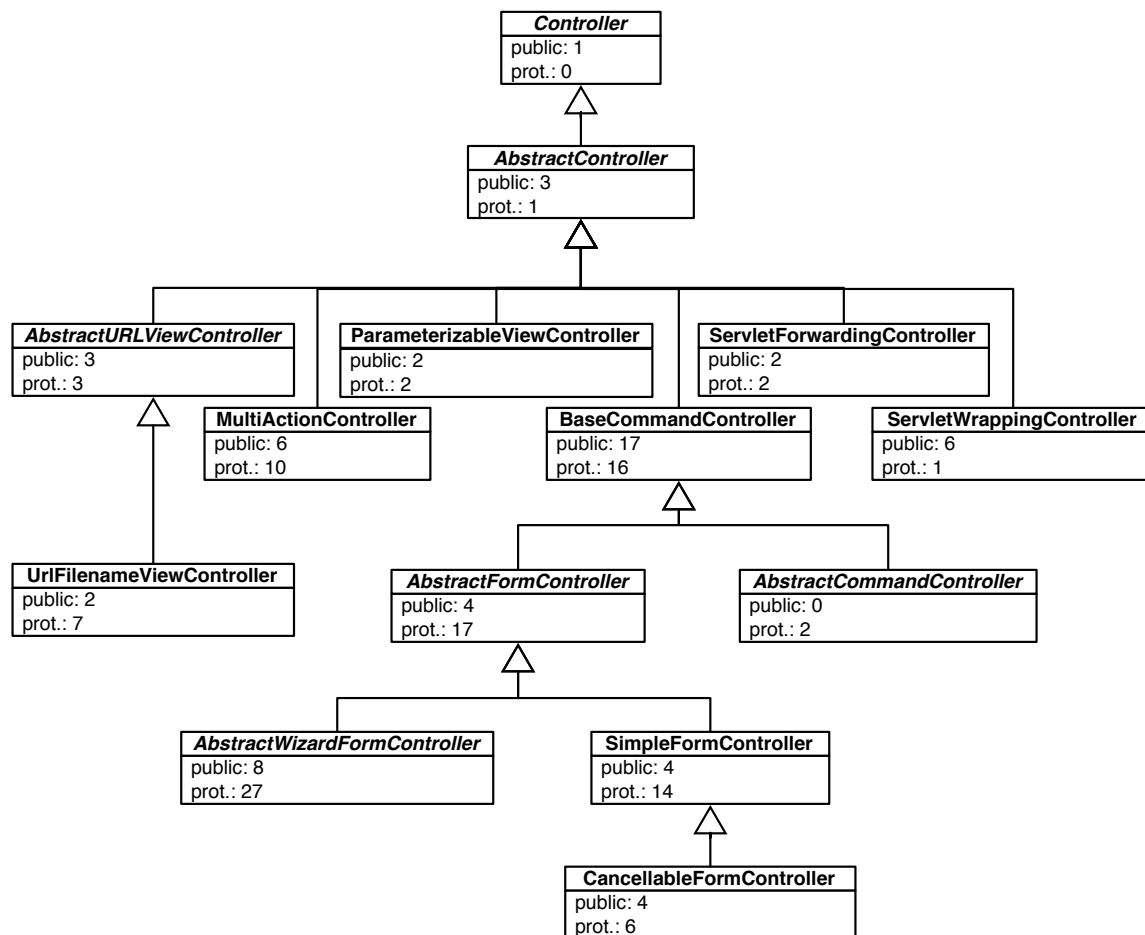
Figure 6.1: UML class diagram of the Spring `Controller` hierarchy. Italicized class names are abstract classes or interfaces. This diagram also lists the number of public and protected methods defined or redefined in each class. The most commonly extended class is generally regarded to be `SimpleFormController`, which is five levels deep in the hierarchy and has access to 77 public or protected methods above it. Some of these are implementations of an abstract method declared higher in the interface or overridden implementations, but most are not.

effectively represents the raw request from the user for a web page. Any further functionality is provided by the leaf classes, like `SimpleFormController`, which is somewhat equivalent to a very simple `Page` in ASP.NET. However, the API of `SimpleFormController` is much more complex as it is spread across this entire hierarchy.

The differences in business drivers have lead to significant differences in the design of these two frameworks. This makes Spring a useful and interesting framework for studying the generalizability of relationship-based specifications.

6.2   Methodology for gathering examples

In Chapter 3, I described a study on the ASP.NET help forums where I went through 271 forum threads that had last activity during a one week time period in October, 2006. From this, I identified 16 threads where a developer had a specific coding problem and received a usable response. I identified the collaboration constraint within these 16 threads and noted the properties that they shared in common.

This same methodology was not effective for gathering examples from the Spring forums. Unlike the ASP.NET forums, the Spring forums have no reward system to encourage the community to answer questions; therefore, there were significant numbers of unanswered questions. Additionally, as Spring is meant to integrate with so many other technologies, there were far more tutorial requests of the form "How do I get Spring to work with X?"

In order to find examples effectively, I created an automatic filtering system that would scan threads for specific properties and only return those that met my criteria. The criteria I used are:

- *Has a* `<pre>` *tag.* To ensure that there was a specific example being discussed and filter out requests for tutorials and documentation, I accepted only threads where there was code posted within an HTML `<pre>` tag (for pre-formatted text, commonly used for displaying code). This might miss threads where people did not use the `<pre>` tag to display code.

- *Uses words "exception" or "error".* Again to filter out requests for tutorials and documentation, I accepted only threads where the words "exception" or "error" appeared somewhere. I found this had a higher false positive rate than I expected, as people were asking for tutorials on how to show an error message. However, it generally seemed to keep only posts where some error occurred in a developer's program. This unfortunately means that I missed any issues where the error was unexpected run time behavior, rather than an exception.

- *Responded to by a top-poster.* I accepted only threads where one of the responders was in the top-25 of all posters. I found that these posters are careful to respond only to problems that they can understand and reproduce, and they are more likely to provide a solution. This filtering mechanism will miss any threads that were correctly solved by a user with a lower post count.

- *Has an affirmation.* To ensure that there actually was a solution presented, I accepted only threads where the *original* poster had a secondary post with one of the following phrases: "solved", "that work", "works", and "working". This is meant to capture threads where the

Table 6.1: Filtering properties applied to the ASP.NET example threads from Table 3.1a in Chapter 3. Nearly all provided code, and about half used the keywords I was looking for. A majority were also responded to by an All-Star or Star level responder, indicating a significant amount of expertise. While few people on the forums directly affirmed a correct solution in a posting, many would come back to check the "solution" box next to the post which solved their problem, indicated with "(Checked)" in the column. Spring does not have this feature on their forum.

Number	Code	Error	All-Star or Star responder	Affirmed
1031123	Y	Error	Y	
1031139	Y	Error, Exception	Y	
1031804	Y			
1032020	Y	Error		Y
1031933	Y			Y
1030504	Y		Y	
1027694			Y	
1032187	Y			(Checked)
1032278	Y	Exception	Y	(Checked)
1032624	Y			
1032991	Y	Error	Y	
1033020	Y	Error	Y	Y
1033046			Y	
1031946	Y	Error	Y	(Checked)
1033217	Y	Error	Y	Y
1033450	Y	Error	Y	(Checked)

original poster returns to say "Thanks! That worked for me." This filter will miss threads with solutions where the original poster either did not return or did not respond in this way.

Additionally, I limited the date range to be before October of 2007 in order to capture only Spring 2.0, as the next version of Spring had significant API changes.[1]

As seen in Table 6.1, these criteria also appeared in my 16 ASP.NET examples. Therefore, while not gathered using the same method, I believe that this technique was a good way to capture the interesting and relevant posts for this case study.

6.3 Quantitative Results

Using the methodology described above, I found 156 threads that met my criteria; all of these are archived [2]. I then determined which of these threads described a violated collaboration constraint, and of those, which were possible to describe using Fusion. As Table 6.2 shows, 53 had collaboration constraints, and 17 of these were specifiable in Fusion. Another 34 would be

---

[1]I chose not to use the newer version of Spring as it heavily uses annotations rather than subtyping to identify callback locations. While it is theoretically possible to use relationships for either one, I have not implemented annotation support in Fusion at this time.

Table 6.2: Breakdown of threads in Spring. There were several features that could be added to make Fusion work for more constraints. JSP is a commonly-used language to describe the view of a Spring webpage, and there were many constraints that need to match JSP code to the XML and Java. OGNL is a language that can be used inside of XML to execute simple expressions; Spring uses it to execute arbitrary code in XML. Two threads were about a collaboration constraint that describes a requirement on the filesystem, though nearly all of the JSP-based collaboration constraints would also need this feature support. Some collaboration constraints required the XML to be aware of an object's fields and methods, so field and reflection support are necessary to handle these. Finally, simple string manipulation, such as handling concatenate, was needed for one thread in addition to many of the JSP, reflection, and file resource threads.

Not a collaboration constraint	103
Requires JSP support	17
Requires OGNL support	4
Requires file resource support	2
Requires field support	5
Requires reflection support	5
String manipulation	1
Broken beyond repair	2
Specifiable in Fusion	17
Total	156

specifiable with additional feature support for Fusion, as detailed in Table 6.2. There were also two threads that had a collaboration constraint, but the posters had so completely mangled their code that I could see no way for Fusion to help them. Most collaboration constraints require that the developer do something correct for the constraint to trigger in the first place. However, these developers appeared to not even be using the right APIs to start with and needed to start over entirely.

The remaining 103 threads were not useful for the study. These contained mostly requests for tutorials, but there were also feature requests, Spring bug reports, issues about associated frameworks (like Acegi Security framework or Hibernate persistence framework). There were also a few postings which might have been collaboration constraints, but there was so little information that I could not even categorize the problem.

Surprisingly, the collaboration constraints described in the 17 threads only spanned eight collaboration constraints, as shown in Table 6.3. Two particularly problematic constraints covered 53% of the threads. Like the examples in ASP.NET, where three constraints covered 63% of the threads, it appears that specifying only a few problematic APIs would provide significant benefit.

Based on the examples from the threads and the solutions given, I created 24 test programs, including good and bad programs for each of the APIs [1]. To keep these programs similar to snippets from a fully functioning web application, I created them by modifying the JPetStore [63] and PhoneBook [99] examples that are distributed with Spring. The examples included the relevant classes containing the error, all referenced classes, and the original XML configuration files. It was important to include these files since, as discussed in Chapter 5, their presence changes the analysis results. For each API, I used as much of the code as possible from the original forum

Table 6.3: Analysis of collaboration constraints found in the Spring threads. I used the same criteria for classification as in Table 3.1b in the ASP.NET study. These threads can be accessed through the URL `http://forum.springsource.org/showthread.php?<NUMBER>` and also are archived at [2].

Numbers	API	#Classes, #Objects	Extrinsic v. Intrinsic	Semantics	Artifact Types
13320, 21751, 33139, 33168, 33456, 36333	OnSubmit	6, 5	Extrinsic	Callback, Identity, Value	Java
26787, 36109, 43182	SetupForm	1, 1	Extrinsic	Callback, Identity, Temporal	XML
32429, 39040	AppContext	3, 2	Extrinsic	Identity, Value	Java, XML
28603, 39209	MAVModel	4, 4	Intrinsic	Callback, Identity	Java
39480	RefData	4, 4	Extrinsic	Callback, Identity, Value	Java, XML
36891	ViewResolver	2, 2	Extrinsic	Temporal, Value	XML
38940	Action	2, 1	Extrinsic	Identity, Value	Java, XML
43643	SerialFlow	2, 1	Extrinsic	Identity, Value	Java, XML

Table 6.4: Complete results from the Spring case study. The first columns give the API name, the section this API is discussed in, and the names of the example programs that I created based upon the forum threads. The "Ideal" column shows what a perfect analysis should give; an "X" represents an error, and a checkmark represents a passing example. The final three columns show the results from the analyses. Results that match the ideal are in bold green font. The full code for the examples is archived in [1].

API (Section)	Example name	Ideal	Sound	Pragmatic	Complete
AppContext (§6.4.1)	Correct	✓	X	✓	✓
AppContext (§6.4.1)	BadFactory	X	X	X	✓
AppContext (§6.4.1)	BadBean	X	X	X	✓
OnSubmit (§6.4.2)	SameViewsCorrect	✓	X	✓	✓
OnSubmit (§6.4.2)	DiffViewsCorrect	✓	X	✓	✓
OnSubmit (§6.4.2)	SameViewsIncorrect	X	X	X	✓
ViewResolver (§6.4.3)	CorrectOnlyOne	✓	X	✓	✓
ViewResolver (§6.4.3)	CorrectChainEnd	✓	X	✓	✓
ViewResolver (§6.4.3)	NotEndOfChain	X	X	X	X
RefData (§6.4.4)	Correct	✓	X	✓	✓
RefData (§6.4.4)	ChangedRequest	X	X	X	X
RefData (§6.4.4)	UsedFBO	X	X	X	X
MAVModel (§A.1)	CorrectWithPOJO	✓	X	✓	✓
MAVModel (§A.1)	CorrectWithMap	✓	X	✓	✓
MAVModel (§A.1)	IncorrectWithMap	X	X	X	X
MAVModel (§A.1)	IncorrectAddingMap	X	X	X	X
Action (§A.2)	CorrectType	✓	X	✓	✓
Action (§A.2)	IncorrectType	X	X	X	X
SerialFlow (§A.3)	CorrectFlow	✓	X	✓	✓
SerialFlow (§A.3)	CorrectNotFlow	✓	X	✓	✓
SerialFlow (§A.3)	IncorrectNotSerial	X	X	X	X
SetupForm (§A.4)	CalledSetupDirect	✓	X	✓	✓
SetupForm (§A.4)	CalledSetupIndirect	✓	X	✓	✓
SetupForm (§A.4)	ForgotSetup	X	X	X	✓

Table 6.5: Summary of results from the Spring case study from Table 6.4. This table compares results of the 24 examples from the three variants to the "ideal" analysis that has no false results. In these examples, the pragmatic variant matched ideal, and the complete variant did surprisingly well. The sound variant was never able to be precise enough to verify a program as correct.

	True Positive (X)	True Negative (✓)	False Positive (X)	False Negative (✓)
Ideal	11	13	0	0
Sound	11	0	13	0
Pragmatic	11	13	0	0
Complete	7	13	0	4

thread and copied it into either JPetStore or PhoneBook to make the "bad" examples. I created the good example by making the change suggested by the responders on the forum threads. I also created additional examples by making some reasonable assumptions of other ways that a developer might break the same constraint.

To test Fusion's ability to detect the errors, I created specifications for each of the eight API's. I then ran the three variants of the analysis; the pragmatic variant was run with the may-like variant of the points-to analysis. The detailed results are displayed in Table 6.4, and a summary is shown in Table 6.5.

As seen in Table 6.4, the pragmatic variant with the shared points-to analysis clearly outshone the competition. While it appears perfect in these examples, it would not likely do as well in large programs with more aliasing possibilities and would begin to act more like the complete analysis. However, for running on examples of the size posted on the forums, it does quite well and arguably would have helped many people find the defect in their programs without using the forums.

Regarding performance, the analysis runs fast enough to not be a concern for small program such as these. The first run of the analysis takes longer as there is a global search through the classpath to create a type hierarchy; while this should theoretically as fast as the compiler, there are several bugs in Eclipse's implementation that cause this to take several minutes to run. Because of this major performance hit, Fusion caches the entire hierarchy for later use. Secondary runs take only a few seconds, as the Fusion analysis itself is very fast. A further discussion of performance and scalability, on a more substantial program, can be found in Chapter 7.

6.4 Detailed Examples

This section will present four specific examples from the case study to better understand the nature of the collaboration constraints that were seen and the extent to which Fusion could specify the constraint. The first two examples are meant to show the expressiveness of Fusion; the first is a small example that is not easy to capture in other specification systems, and the second is a larger example that uses nearly all of the expressiveness of Fusion. The next two examples are interesting because they made explicit some of the tradeoffs that occur in a specification language as abstract as Fusion and show its flexibility to meet the needs of the specification writer. The remaining examples were similar in nature to those in this section, and brief descriptions of the problems alongside the Fusion specifications for them can be found in Appendix A.

6.4.1 Object identity

In previous chapters, I have described object identity as an important aspect of collaboration constraints, and it is one which is not easily capturable using many existing specifications systems, as discussed in Chapter 8. The Spring forums provide an example that showcases how object identity is an integral part of interacting with modern frameworks like Spring.

Like many other frameworks, Spring uses dependency injection to automatically wire together components from a declarative file [42]. In Spring, the developer creates new objects by declaring them in a `<bean>` tag in a Spring configuration file. Listing 6.1 shows example tags; Spring

Figure 6.2: Class diagram of the `ApplicationContext` and `ApplicationObjectSupport`.

Listing 6.1: An example of dependency injection in Spring.

```
1   <beans>
2     <bean id="accountValidator" class="AccountValidator"/>
3
4     <bean id="petDatabase" class="Database">
5       <property name="user" value="foo"/>
6       <property name="databaseName" value="pets"/>
7     </bean>
8
9     <bean id="myStore" class="PetStore">
10      <property name="database" ref="petDatabase"/>
11    </bean>
12
13    <bean id="accountController" class="AccountFormController">
14     <property name="petStore" ref="myStore"/>
15     <property name="validator" ref="accountValidator"/>
16    </bean>
17  </beans>
```

will use this information to create four objects with the type specified. Based on this file, Spring will set the `database` field of `PetStore` object to be the object declared as `petDatabase`, and the `AccountFormController` that is created will reference both the `AccountValidator` and the `PetStore` objects. Spring uses reflection and setter methods to provide this functionality.

In Spring, dependency injection is used for many things, but one of the most important is injecting the *application context*. The application context represents the collection of bean objects that Spring instantiated together from the same configuration file, and it is concretely represented with the `ApplicationContext` type. The `ApplicationContext` interface, seen in Figure 6.2, has a one method of interest for our purposes: `Object getBean(String beanName)`. This method will return the unique object represented by the given name in the configuration file. For example, we can call `ac.getBean("myStore")` to get the `PetStore` object that is represented in Listing 6.1. The `ApplicationContext` itself is injected into any bean which extends `ApplicationObjectSupport`; this class has a single setter/getter pair to inject and retrieve the `ApplicationContext`.

In thread number 32429 [84], these two simple interfaces cause a problem for the user "pom-

piuses", who posts about a null pointer problem he is having. He is helped by Marten Deinum, a Spring expert who is frequently on the forums. A shortened version of the exchange between then, shown below, is quite interesting.

> **pompiuses:** *If I extend* `ApplicationObjectSupport`*, I should according to documentation be able to get the applicationContext using the method* `getApplicationContext()`*.*
>
> *The problem is that it always returns null no matter what. What I'm I missing here??*
>
> *I know for a fact the the applicationContext is not null, because if I i.e extend* `Abstract-Controller` *in one of my controllers and then use the* `getApplicationContext()`*method, it works.*
>
> **Marten Deinum:** *How are you instantiating the object extending* `ApplicationObject-Support`*. It implements the* `ApplicationContextAware` *interface so the* `Application-Context` *should be automatically injected if specified/configured inside a applicationContext file.*
>
> **pompiuses:** *I instantiate it like any other object ;*
> `MyObject myObject = new MyObject();`
>
> *Exactly what needs to be specified inside a applicationContext file? MyObject?*
>
> **Marten Deinum:** *When you create an object with new it isn't a Spring managed bean and hence not being injected with anything or under Spring management. Assuming you already running some kind of application you already have a applicationContext.xml (or whatever the name is you specified). For more information check the first few chapters of the Spring reference guide.*
>
> *Configure your bean as a prototype and retrieve instances from the applicationcontext.*
> `<bean id="myObject" class="MyObject" scope="prototype"/>`
>
> *Then from some other spring managed bean*
> `MyObject object = (MyObject) context.getBean("myObject");`
>
> **pompiuses:** *Yes I know I can create a bean in the application context and retrieve it the way you describe, but that's not the issue here.*
>
> *As I wrote,* `MyObject` *extends* `ApplicationObjectSupport`*. That should enable* `MyObject` *to access the* `ApplicationContext` *using the* `getApplicationContext()` *method.*
>
> *I want this because then I can fetch beans, using* `applicationContext.getbean("some-Bean")`*, from* `MyObject`*.*
>
> *But since* `getApplicationContext()` *always returns null, something is not right.*
>
> **Marten Deinum:** *Wel actually it is*
>
> *First of all if you want to have the applicationContext injected it MUST be a spring managed bean. If it isn't your* `ApplicationContext` *isn't going to be injected. So object created with new* `SomeObject` *implementing* `ApplicationContextAware` *are never going to be injected with the applicationcontext....*

> (This is followed by a detailed one-page explanation about the internals of how this works.)
>
> **pompiuses:** *Thanks for the great input! I got it working by adding this line into my applicationContext.xml:*
> ```
> <bean id="myObjectBean" class="com.something.MyObject"/>
> ```

In this exchange, Marten quickly guessed, and confirmed, the root of the problem: the poster was creating objects with `new`, rather than allowing them to be "Spring-managed" by creating them in the XML configuration file. Even with an expert, pompiuses requires two explanations in order to understand these fairly simple interface. This user was not fully aware of how the object's *identity*, not its type, is responsible for whether `getApplicationContext()` returns null.

Rather than a page of English text, I'll specify the constraint using a few Fusion specifications. To represent an object that is Spring-managed, there will be a relationship

$$Context(String, Object, ApplicationContext)$$

where the first parameter is the unique name of a bean from the configuration file, the second parameter is the bean itself, and the third parameter is the application context that manages the bean. This single relationship will allow us to specify both of the constraints surrounding `ApplicationContext`.

First, to get an `ApplicationContext`, the `ApplicationObjectSupport` object that we have must already be managed by an `ApplicationContext`. The constraint for this is simple:

```
1  @Constraint(
2      op="ApplicationObjectSupport.getApplicationContext() : ApplicationContext",
3      restrictTo="Context(name, target, result)",
4      requires="Context(name, target, result)"
5      )
```

That is, we restrict this call to only return an object for which a Context exists, and we require that such a Context actually exists.

The second constraint is that when we have an `ApplicationContext`, all requests to get a bean must be valid. As it turns out, this constraint has identical form to the one above.

```
1  @Constraint(
2      op="ApplicationContext.getBean(String name) : Object",
3      restrictTo="Context(name, result, target)",
4      requires="Context(name, result, target)"
5      )
```

Of course, for these constraints to work, we must have prior knowledge about the Context relationships that exist from the XML configuration files. Listing 6.2 provides the XQuery that makes this happen.

What is particularly interesting about this example is that a type-based approach cannot capture unique identities of objects, yet only a few specifications and a single relationship can specify this problem. This example could be further improved if Fusion was aware of the file-system resources; this would allow Fusion to properly handle the case where a single application context loads beans from two or more XML files. In its current state, Fusion will treat these as separate application contexts.

Listing 6.2: XQuery to retrieve the relationship Context from a Spring configuration file

```
1  declare namespace sf="http://www.springframework.org/schema/beans";
2  declare variable $doc as xs:string external;
3
4  for $bean in doc($doc)/sf:beans/sf:bean
5  return <Relationship name="Context" effect="ADD">
6          <Object name ="{data($bean/@id)}" type="java.lang.String"/>
7          <Object name ="{data($bean/@id)}" type="{data($bean/@class)}"/>
8          <Object name ="{$doc}" type="org.springframework.context.ApplicationContext"/>
9      </Relationship>
```


Figure 6.3: A diagram showing the data flow from a user's browser through the Spring framework and back to the user as HTML.

6.4.2   Expressiveness for complex constraints

In this section, I present an API that is both difficult to use (6 threads referenced this API, as seen in Table 6.3) and which fully exercises the expressiveness of the specification language. The example comes from the `SimpleFormController` class, perhaps one of the most commonly used classes of the Spring MVC framework. This API is discussed in all the popular books on Spring [51, 61, 113] and included in the official tutorial on the MVC framework [90], yet it is still an API that is easy to break in many ways.

It is best to first understand how the API is used in *most* situations. At a high level, the interaction between the end-user and the Spring MVC components is as shown in Figure 6.3. The end-user requests a web page containing a form using an HTTP GET request. Spring looks up the `Controller` for this request and passes the request on to this `Controller`. The controller will return a `ModelAndView` object back to the Spring framework; this object contains the name of a view and a `Map` of the model data that the view might need. The Spring framework then finds the view (likely a JSP page), passes it the model data, and returns HTML to the user.

When the user enters data into their browser and clicks the submit button, the browser sends

Listing 6.3: A simple form to edit an account

```
 1  public class EditAccountForm extends SimpleFormController {
 2    private Database db;
 3
 4    public void setDatabase(AccountDatabase db) {this.db = db;}
 5
 6    public Object formBackingObject(HttpServletRequest request) throws Exception {
 7      Integer id = request.getAttribute("accountID");
 8      if (id == null || id.intValue() <= 0)
 9        throw new AccountException("Can only edit accounts with an id greater than 0")
10      return db.getAccount(id.getInteger());
11    }
12
13    public Map referenceData(HttpServletRequest request) throws Exception {
14      Map data = new HashMap();
15      data.put("states", db.getAllStates());
16      data.put("countries", db.getAllCountries());
17      return data;
18    }
19
20    public ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response,
21                          Object command, BindException errors) throws Exception {
22      Account account = (Account)command;
23      db.save(account);
24      return new ModelAndView(getSuccessView(), null);
25    }
26  }
```

an HTTP POST message to the Spring framework with the user's data attached. The POST process happens nearly the same way as the GET process. The only difference might be that the `Controller` stores the user data submitted and likely returns a different model and view for the user to move on to (ie: a "Thank you for submitting!" page).

The purpose of the `SimpleFormController` is to encapsulate much of this for reuse. Developers can extend from `SimpleFormController` to easily create a simple form with a single submit button and can override key methods to get basic functionality. For example, Listing 6.3 provides an implementation for a form to edit account information. The method `formBackingObject` returns an object that represents the initial data to show to the user (the existing account in the database). The method `referenceData` returns a `Map` of all data that is relevant to the form, but is not part of an individual submission (like the list of states and countries). Finally, the method `onSubmit` stores the data to the database and sends the user to a "success" page to confirm that their account change was saved.

The last step necessary to make this work is the XML configuration file, seen in Listing 6.4. As seen, this creates an instance of the class in Listing 6.3 with a particular form view and success view. The command name will match the command name used in the form view JSP, and that view will expect an object with the type given by command type. The command type is also the same as the type returned by `formBackingObject`. As given in Listings 6.3 and 6.4, this form will

Listing 6.4: Configuration for an edit account form

```
1  <bean id="editAccountForm" class="AccountFormController">
2    <property name="database" ref="myAccountDatabase"/>
3    <property name="formView" value="editAccount"/>
4    <property name="successView" value="thanks"/>
5    <property name="commandName" value="accountForm"/>
6    <property name="commandType" value="Account"/>
7  </bean>
```

work as expected.

Now, we will add a seemingly minor twist. Instead of returning to a thank you page, let's say our developer wants to go back to the same form. Therefore, in Listing 6.4, she changes the success view as follows:

```
4      <property name="successView" value="editAccount"/>
```

Our developer isn't entirely naive; she knows that in order to go to the form view, she'll need to provide the appropriate model data. Therefore, she also changes the return from the `onSubmit` method to return the user's entered data.[2]

```
20  public ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response,
21                     Object command, BindException errors) throws Exception {
22    Account account = (Account)command;
23    accounts.save(account);
24     return new ModelAndView(getSuccessView(), errors.getModel());
25  }
```

She runs her application, and at first, everything appears fine. She can enter data on her form, she can submit it, and it sends her back to the form again, with *almost* all of her data in place. Her text boxes all have data, but the drop down lists for the states and countries are completely empty!

As it turns out, when an HTTP POST occurs to the `SimpleFormController`, it will bind the user's data into `errors.getModel()`, but it *won't* call `referenceData` and bind that as well. Presumably, this is because the reference data won't be needed for the success view. Of course, this isn't the case when the success view happens to be the form view.

There are two ways to solve this problem. The first is to manually call `referenceData` and store the result into the model map, but this is not recommended. The recommended practice is to instead return from `onSubmit` with a call to `showForm`, as shown:

```
20  public ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response,
21                     Object command, BindException errors) throws Exception {
22    Account account = (Account)command;
23    accounts.save(account);
24     return showForm(request, response, errors);
25  }
```

---

[2]Don't ask why the model data is stored in an object called `errors` with type `BindException`. It's not my design choice, nor is it relevant to the problem, and the answer may be longer than this thesis. Just go with it.

This call is made automatically when an HTTP GET occurs, but not on HTTP POST. In addition to setting up the reference data, it also does several other tasks necessary for proper form functionality.

This constraint is very simple to trigger (all we need is for the success view to be the same as the form view), yet very difficult to discover and fix. Of course, by specifying this constraint in Fusion, we will help the developer to find the problem before she runs her application and walks through all the steps necessary to trigger the problem.

To specify this constraint in Fusion, we will need four relationship types:

1. FormViewName(SimpleFormController, String) represents the association between a `SimpleFormController` and the string id of its form view.

2. SuccessViewName(SimpleFormController, String) represents the association between a `SimpleFormController` and the string id of its success view.

3. MAVViewName(ModelAndView, String) represents the relationship between a `ModelAndView` object and the string id of the view it contains.

4. ShowForm(ModelAndView, HttpServletRequest, BindException) represents the relationship between a `HttpServletRequest` and a `BindException` when they are used as parameters to a `showForm` call and return the given `ModelAndView`.

The only relationships retrieved from XML are the FormViewName and SuccessViewName relationships; the XQuery to retrieve these is shown in Listing 6.5.

The specifications for the constraint on how to return from `SimpleFormController.onSubmit` are in Listing 6.6. The first two specifications are straightforward: upon requesting either a form view or a success view, Fusion will restrict the possible options for the return value to be only what was already known from the configuration file. The next two are also straightforward, as they simply associate a `ModelAndView` object with the view parameter that was used at its construction with the MAVViewName relationship. The next specification is more interesting; the goal here is to find out that the returned `ModelAndView` from a call to `showForm` always will have the form view as its view. Since we already have the FormViewName relationship and wish to use the one we have, this relationship appears in the trigger predicate. This will then bind the `view` parameter to the appropriate object when we create the MAVViewName relationship later.

Finally, the constraint itself is at the end of the `onSubmit` method, specified with the operation `EOM: SimpleFormController.onSubmit`. Enforcing the desired rule is now simple. We are only concerned with the case where we are attempting to return a `ModelAndView` object from this method, and that `ModelAndView` object's view is our form view. In this case, we require that `ModelAndView` *must* have been the result of a proper call to `showForm`.

As seen in Table 6.4, this constraint works exactly as expected with the pragmatic variant. In the original example, where the success view and form view are different, the final constraint won't trigger because the view of the `ModelAndView` being returned is not a form view. However, if it is a form view, then it will ensure that this `ModelAndView` object was the result of a call to `showForm`, as opposed to a call to `new ModelAndView`.

Listing 6.5: Retrieve the relationships FormViewName and SuccessViewName from a Spring XML file

```
1  declare namespace sf="http://www.springframework.org/schema/beans";
2  declare namespace fusion="http://code.google.com/p/fusion";
3  declare variable $doc as xs:string external;
4
5  for $bean in doc($doc)/sf:beans/sf:bean
6  let $formView := $bean/sf:property[@name="formView"]
7  where fusion:isSubtype($bean/@class,"org.springframework.web.servlet.mvc.SimpleFormController")
8      and not(empty($formView))
9  return <Relationship name="FormViewName" effect="ADD">
10         <Object name ="{data($bean/@id)}" type="{data($bean/@class)}"/>
11         <Object name ="{data($formView/@value)}" type="java.lang.String"/>
12     </Relationship>
13
14 for $bean in doc($doc)/sf:beans/sf:bean
15 let $successView := $bean/sf:property[@name="successView"]
16 where fusion:isSubtype($bean/@class,"org.springframework.web.servlet.mvc.SimpleFormController")
17     and not(empty($successView))
18 return <Relationship name="SuccessViewName" effect="ADD">
19         <Object name ="{data($bean/@id)}" type="{data($bean/@class)}"/>
20         <Object name ="{data($successView/@value)}" type="java.lang.String"/>
21     </Relationship>
```

6.4.3 Trigger predicate v. Requires predicate

The next example highlights how the pragmatic variant is affected by the form of the specifications. In particular, we will see two specifications that, while identical within the sound and complete variants, are different under the pragmatic variant due to how the pragmatic variant treats the trigger and requires predicates differently.

This example will study the use of `ViewResolvers` in Spring. As seen in the last section, `Controllers` return a `ModelAndView` object which contains the name of a view. A `ViewResolver` looks up this name, retrieves a file on the system, and does any processing to associate the model with the view. For example, the `InternalResourceViewResolver` in Listing 6.7 will look up a JSP file and use the model data as the parameters to the JSP. After processing, the resulting data is sent back to the end-user that made the original HTTP Request.

In Spring, a `ViewResolver` may handle HTML, JSP, TXT, or even a PDF. To deal with all of these within a single application, Spring allows a programmer to chain `ViewResolvers` together so that if the first one in the chain cannot find the view that goes with the identifier, it can pass the request on to the next `ViewResolver`. However, some `ViewResolvers` don't forward the request through; these `ViewResolvers` can only be the last item in the chain. The `InternalResourceViewResolver` is one such example; if it cannot find the view for an identifier, it will simply return with no view. In fact, all subtypes of `UrlBasedViewResolver`, of which `InternalResourceViewResolver` is one, will not forward a request and must be last in the chain.

This can cause confusion, as was the case for the programmer "ilpata" in thread number 36891 [56] from Table 6.3. This programmer was attempting to use two `ViewResolvers` but chained them

Listing 6.6: Specifications for the correct return from `SimpleFormController.onSubmit`

```
1   @Constraint(
2       op="SimpleFormController.getFormView() : String",
3       restrictTo="FormViewName(target, result)"
4   )
5   @Constraint(
6       op="SimpleFormController.getSuccessView() : String",
7       restrictTo="SuccessViewName(target, result)"
8   )
9
10  @Constraint(
11      op="ModelAndView(String view)",
12      effect={"MAVViewName(result, view)"}
13  )
14  @Constraint(
15      op="ModelAndView(String view, Map model)",
16      effect={"MAVViewName(result, view)"}
17  )
18
19  @Constraint(
20      op="SimpleFormController.showForm(HttpServletRequest request, HttpServletResponse response,
21              BindException errors) : ModelAndView",
22      trigger="FormViewName(target, view)",
23      effect={"MAVViewName(result, view)", "ShowForm(result, request, errors)"}
24  )
25
26  @Constraint(
27      op="EOM: SimpleFormController.onSubmit(HttpServletRequest request, HttpServletResponse response,
28              Object command, BindException errors) : ModelAndView",
29      trigger="MAVViewName(result, view) AND FormViewName(target, view)",
30      requires="ShowForm(result, request, errors)"
31  )
```

Listing 6.7: Incorrect resolver chain

```
1   <beans>
2     <bean id="jspViewResolver"
3         class="org.springframework.web.servlet.view.InternalResourceViewResolver">
4       <property name="order" value="1"/>
5       <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
6       <property name="prefix" value="/WEB-INF/jsp/"/>
7       <property name="suffix" value=".jsp"/>
8       </bean>
9
10    <bean id="alternativeViewResolver"
11        class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
12      <property name="order" value="2"/>
13      <property name="basename" value="views"/>
14    </bean>
15  </beans>
```

Listing 6.8: XQuery to retrieve the relationship ResolverChain from a Spring configuration file

```
1  declare namespace sf="http://www.springframework.org/schema/beans";
2  declare namespace fusion="http://code.google.com/p/fusion"
3  declare variable $doc as xs:string external;
4
5  for $res1 in doc($doc)/sf:beans/sf:bean
6  for $res2 in doc($doc)/sf:beans/sf:bean
7  where fusion:isSubtype($res1/@class, "ViewResolver") and
8      fusion:isSubtype($res2/@class, "ViewResolver") and
9      $res1/sf:property[@name="order"]/@value = ($res2/sf:property[@name="order"]/@value - 1)
10 return <Relationship name="ResolverChain" effect="ADD">
11     <Object name ="{data($res1/@id)}" type="{data($res1/@class)}"/>
12     <Object name ="{data($res2/@id)}" type="{data($res2/@class)}"/>
13   </Relationship>
```

so that the `InternalResourceViewResolver` was first rather than last, as seen by the configuration file posted in Listing 6.7. This programmer was particularly confused because there was a secondary bug that would cause the `ResourceBundleViewResolver` to fail if it was ever run, so from "ilpata"'s perspective, it was at least partially working when the `InternalResourceViewResolver` was first in the chain. Due to the delayed nature of the error when the `InternalResourceView-Resolver` is first in the chain, "ilpata" assumed that this was more correct than the opposite and so had to be told three times by the experts that this was the primary issue and that a secondary issue was causing the other error.[3] Because of this, the thread took three days to resolve.

By specifying this in Fusion, we can detect the defect at compile time, and hopefully make it clear to "ilpata" earlier that the chaining issue is the primary problem. To create the constraint, I use the relation

$$ResolverChain(ViewResolver, ViewResolver)$$

to describes a chain of two resolvers where the second parameter comes after the first parameter in the chain. A larger chain of size n can then be represented by $n - 1$ ResolverChain relationships. The XQuery in Listing 6.8 will retrieve these relationships from a Spring XML file; thus, the XML from Listing 6.7 will produce the single relationship ResolverChain(jspViewResolver, alternativeViewResolver).

The constraint itself seems fairly straightforward. As this constraint only concerns XML, and not Java, we will use the "XML" operator in Fusion to verify that the XML passes the constraint right after all XML files have been processed by the XQuery. At this point, if we have an object of type `UrlBasedViewResolver`, we must ensure that it does not have anything after it in the chain. This could be written as:

```
1  @Constraint(
2      op="XML",
3      trigger="prevRev instanceof UrlBasedViewResolver",
4      requires="!ResolverChain(prevRev, nextRev)"
5  )
```

---

[3]The secondary issue is not currently specifiable by Fusion, as it requires knowledge of URLs and resources.

Listing 6.9: Correct resolver chain of three resolvers

```
1  <beans>
2    <bean id="primaryViewResolver" class="org.springframework.web.servlet.view.XMLViewResolver">
3        <property name="order" value="1"/>
4    </bean>
5
6    <bean id="alternativeViewResolver"
7          class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
8      <property name="order" value="2"/>
9      <property name="basename" value="views"/>
10    </bean>
11
12   <bean id="jspViewResolver"
13         class="org.springframework.web.servlet.view.InternalResourceViewResolver">
14     <property name="order" value="3"/>
15     <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
16     <property name="prefix" value="/WEB-INF/jsp/"/>
17     <property name="suffix" value=".jsp"/>
18   </bean>
19 </beans>
```

The constraint above will allow all three variants to detect the error in Listing 6.7. However, when this constraint is used on correct code, such as that in Listing 6.9, something interesting occurs: the pragmatic variant believes there is still a bug. A little investigation reveals the source: while the trigger predicate is True, the requires predicate is Unknown. While we do not have any ResolverChain relationships with `jspViewResolver` as the first parameter, we don't know that those relationships are false, either. Our XQuery only created relationships; it did not specify the non existence of the relationships ResolverChain(jspViewResolver, primaryViewResolver) and ResolverChain(jspViewResolver, alternativeViewResolver).

There are two ways to address this issue. The first would be to modify the XQuery to specify non-existence of all other possible relationships. This will have the side effect of also increasing the precision of the sound and complete variants, but the specification cost is high and the analysis run time will be high as well. The second is a seemingly innocuous change: swap the trigger and requires predicate to be a logically equivalent constraint of the form:

```
1  @Constraint(
2      op="XML",
3      trigger="ResolverChain(prevRev, nextRev)",
4      requires="prevRev !instanceof UrlBasedViewResolver"
5  )
```

This works because now the relationship which can produce Unknown is in the trigger clause and the `instanceof` predicate, which only evaluates to True or False, is in the requires clause.

This should seem amiss to the reader: up until this point, we have thought of the association between trigger and requires to be implication. That is, $P_{trg} \implies P_{req}$. However, we have just determined that, for the pragmatic variant, $A \implies \neg B$ is not equivalent to $B \implies \neg A$! In fact, as seen by Table 6.6, these are also not equivalent in the pragmatic variant to $A \wedge B \implies$ False.

While all three forms are logically equivalent[4], and do produce the same results within the sound and complete variants, the pragmatic variant treats them differently.

This is a core feature of the pragmatic variant's heuristic. The pragmatic variant assumes that if there is enough knowledge for the trigger predicate to be known, then there must be enough for the requires predicate. While this works well in instances where there is no negation, it can cause interesting results when there is negation in the requires predicate, as most constraints and XQuery do not remove relationships explicitly. Unfortunately, there is no hard rule for how to use negation in the requires predicate, and how to write the specification depends on the desired results as shown in Table 6.6. Luckily, using negation seems to be an uncommon paradigm in practice; only this constraint and the constraint from Vignette 3.1 use negation, and the constraints in Vignette 3.1 do explicitly remove the relationship in question, thus avoiding the entire problem..

This constraint highlights how the specification writer's choices make large effects on the analysis results, even on small, well defined constraints. The benefit of Fusion is that it uses heuristics about how a developer might typically write a specification in order to achieve cost-effective results. The entire purpose of the pragmatic variant is to encapsulate a heuristic that triggers are intended to be true, rather than unknown. While such heuristics can backfire, they generally provide better results than either a provably sound or provably complete system, as seen in the results from Table 6.4.

6.4.4 Objects v. Operations

The final example explores the tradeoffs that can occur between the complexity of the specification and the precision of the analysis. As it will turn out, more complex and precise specifications are not necessarily better!

Recall that `SimpleFormController.referenceData` should return a `Map` that maps `Strings` to `Objects` for the view to use. This map will contain any data needed for the view, with the exception of the form backing object. Therefore, most implementations of `referenceData` take the following steps:

1. Create a `Map`

2. Get values out of the `Request`

3. Use above values to retrieve data from elsewhere, like a database

4. Put data into the `Map` using predetermined `String` constants that match the variables used in the associated view

As simple as this sounds, the user "CuriousHARD" ran into problems with this when the form kept resetting the user's data. After posting for help on the forums [29], the user "Marten Deinum" found several problems in CuriousHARD's code, including two related to the `referenceData` method displayed in Listing 6.10.

---

[4]They are actually not equivalent when there are variables bound by one and not by the other. While this happens to be the case here (recall the two quantifiers from Chapter 5), it is a secondary issue. The phenomenon described on the pragmatic variant will even arise when a single quantifier works over both A and B.

	A	True	True	True	False	False	False	Unknown	Unknown	Unknown
	B	True	False	Unknown	True	False	Unknown	True	False	Unknown
Sound	$P_{trg} = A$ $P_{req} = \neg B$	Error		Error				Error		Error
Sound	$P_{trg} = B$ $P_{req} = \neg A$	Error		Error				Error		Error
Sound	$P_{trg} = A \wedge B$ $P_{req} = $ **False**	Error		Error				Error		Error
Complete	$P_{trg} = A$ $P_{req} = \neg B$	Error								
Complete	$P_{trg} = B$ $P_{req} = \neg A$	Error								
Complete	$P_{trg} = A \wedge B$ $P_{req} = $ **False**	Error								
Pragmatic	$P_{trg} = A$ $P_{req} = \neg B$	Error		Error						
Pragmatic	$P_{trg} = B$ $P_{req} = \neg A$	Error						Error		
Pragmatic	$P_{trg} = A \wedge B$ $P_{req} = $ **False**	Error								

Table 6.6: A truth table comparing logically equivalent constraints. These three constraints are logically equivalent if we think of a constraint as $P_{trg} \implies P_{req}$, and they are treated equivalently by the sound and complete variants. However, the pragmatic variant treats them differently.

Listing 6.10: Original buggy implementation of `referenceData`, as posted by CuriousHARD [29]

```
1  public class QuotationCntrl extends AbstractWizardFormController {
2    protected Map referenceData(HttpServletRequest request, Object command,
3                         Errors errors, int page) throws Exception {
4      model = new HashMap<String, Object>();
5      model.put("quotation", formBackingObject(request));
6
7      if (page == 0)
8        request.setAttribute("branches", dao.getBranches());
9      else if (page == 1)
10       request.setAttribute("vh", dao.getVehicleDescription());
11     else if (page == 2) {
12       request.setAttribute("policyCoverTypes", dao.getPolicyCoverTypes());
13       request.setAttribute("companies", dao.getInsuranceCompanies());
14     }
15     return model;
16   }
17   ...
18 }
```

1. The first problem is that the code in Listing 6.10 directly manipulates the `request` object. This makes this code fragile, as there is no guarantee that this object's data will be propagated throughout the system; it is given as a parameter for reading data, not for writing data. As seen in Listing 6.11, the correct way to set the values is to create and manipulate a `Map` that is returned from this method and use the `request` as a read-only structure.

2. The second problem is on line 5 of Listing 6.10, where the code actually puts the form backing object into the returned `Map`. As the form backing object is handled separately by the framework, it should not be put into this `Map`, as can be seen in Listing 6.11. Doing so caused the problem seen by CuriousHARD, where the form kept overwriting the user's data with a new form backing object.

Notice that both of these constraints are extrinsic (they constrain operations `HttpServlet-Request` and `Map` respectively), and they only make this constraint within the context of a call to `referenceData`. Therefore, we will use a @Callback specification to signal whether we are within a `referenceData` method. As it turns out, there are actually four such methods in the `AbstractFormController` hierarchy, so we specify all of them as shown in Listing 6.12.[5] The unary relationship used for this callback has type RefData(AbstractFormController).

---

[5]This is not unusual in Spring: there are four versions of the `showForm` method and three versions of the `onSubmit` method. For simplicity, I elided these multiple versions in the earlier example.

Listing 6.11: Correct version of `referenceData`, as posted by Marten Deinum [29]

```
 1  public class QuotationCntrl extends AbstractWizardFormController {
 2    protected Map referenceData(HttpServletRequest request, Object command,
 3                           Errors errors, int page) throws Exception {
 4      model = new HashMap<String, Object>();
 5
 6      if (page == 0)
 7        map.put("branches", dao.getBranches());
 8      else if (page == 1)
 9        map.put("vh", dao.getVehicleDescription());
10      else if (page == 2) {
11        map.put("policyCoverTypes", dao.getPolicyCoverTypes());
12        map.put("companies", dao.getInsuranceCompanies());
13      }
14      return model;
15    }
16    ...
17  }
```

Listing 6.12: Callback specifications on all the versions of `referenceData`.

```
 1  public class AbstractFormController extends BaseCommandController {
 2   @Callback("RefData")
 3    protected Map referenceData(HttpServletRequest request, Object command,
 4                           Errors errors) throws Exception {...}
 5    ...
 6  }
 7
 8  public class SimpleFormController extends AbstractFormController {
 9   @Callback("RefData")
10    protected Map referenceData(HttpServletRequest request) throws Exception {...}
11     ...
12  }
13
14  public class AbstractWizardFormController extends AbstractFormController {
15   @Callback("RefData")
16    protected Map referenceData(HttpServletRequest req, int page) throws Exception {...}
17
18   @Callback("RefData")
19    protected Map referenceData(HttpServletRequest req, Object command,
20                     Errors errors, int page) throws Exception {...}
21     ...
22  }
```

I'll now provide specifications for the first constraint. At the simplest level, we want to prevent calls to `request.setAttribute` from within `referenceData`. This can be accomplished with the following specification:

```
1  @Constraint(
2      op="ServletRequest.setAttribute(String str, Object obj) : void",
3      trigger="RefData(ctrlr)",
4      requires="FALSE"
5  )
```

However, the specification above might be overly general. Is it really the case that we want to prevent *all* calls to this method, on *all* request objects? What we really want is to prevent modification to only the request object used as a parameter into `referenceData`. By abstracting the read-only state of this parameter into a relationship, we can do this instead:

```
1   @Constraint(
2       op="BOM: AbstractFormController.referenceData(HttpServletRequest req, Object command,
3                   Errors errors) : Map",
4       effect={"ReadOnly(req)"}
5   )
6   @Constraint(
7       op="BOM: SimpleFormController.referenceData(HttpServletRequest req) : Map",
8       effect={"ReadOnly(req)"}
9   )
10  @Constraint(
11      op="BOM: AbstractWizardFormController.referenceData(HttpServletRequest req, int page) : Map",
12      effect={"ReadOnly(req)"}
13  )
14  @Constraint(
15      op="BOM: AbstractWizardFormController.referenceData(HttpServletRequest req, Object command,
16   Errors errors, int page) : Map",
17      effect={"ReadOnly(req)"}
18  )
19  @Constraint(
20      op="HttpServletRequest.setAttribute(String str, Object obj) : void",
21      trigger="TRUE",
22      requires="!ReadOnly(target)"
23  )
```

In this specification, the system will mark the parameter as read-only in lines 1-18, and so it will disallow any method calls that are marked as not being read only, such as in lines 19-23. However, writable methods could be called on other `HttpServletRequest` objects, if we had access to any.

These two sets of specifications show how to trade off generality with regard to the object and to the operations. The first set limits a specific operation on all objects, while the second set limits all modifying operations on a specific object. The second set is also more modular and modifiable: if a developer adds new operations to `HttpServletRequest`, she does not need to be aware of all the possible specifications that clients have already written regarding modifiability. Instead, she can just make a constraint similar to lines 19-23 if the new operations is a modifying operation.

From this, we might presume the second set is clearly better to use: it's more precise and more modular. However, there is still an interesting argument for using the first specification: it is small,

Listing 6.13: Retrieve the relationship FormCommand from a Spring XML file

```
1  declare namespace sf="http://www.springframework.org/schema/beans";
2  declare namespace fusion="http://code.google.com/p/fusion";
3  declare variable $doc as xs:string external;
4
5  for $bean in doc($doc)/sf:beans/sf:bean
6  let $cmdClass := $bean/sf:property[@name="commandClass"]
7  let $cmdName := $bean/sf:property[@name="commandName"]
8  let $beanType := data($bean/@class)
9  where fusion:isSubtype($beanType,"BaseCommandController") and not(empty($cmdClass))
10 return <Relationship name="FormCommand" effect="ADD">
11         <Object name ="{data($cmdClass/@value)}" type="java.lang.Class"/>
12         <Object name ="{data($cmdName/@value)}" type="java.lang.String"/>
13         <Object name ="{data($bean/@id)}" type="{$beanType}"/>
14     </Relationship>
```

easy to write and understand, and it will still likely capture most problems with few false posi-
tives. To even get a false positive, we would need access to a second object with the same type, and
that seems unlikely. Likewise, while it isn't flexible to future changes to the `HttpServletRequest`
API, we can rightly question how likely it is for such changes to occur and affect this type of
program.

The second problem from the thread contains a similar tradeoff. Recall that the rule is that we
cannot insert the form backing object into the `Map` returned by `referenceData`. In particular, we
are not allowed to use the command that will be associated with this backing object as a key in
the `Map`. For this constraint, we will the FormCommand(Class, String, BaseCommandController)
relationship.[6] This relationship associates a `BaseCommandController` with the command name
and the class of the backing object that was declared in the XML file; the XQuery to retrieve this
relationship is in Listing 6.13.

Our first attempt at this is simple: prevent all calls to `Map.put` when we are in `referenceData`
and the key matches the command name:

```
1  @Constraint(
2      op="Map.put(String str, Object obj) : Object",
3      trigger="FormCommand(clss, str, ctrlr) AND RefData(ctrlr)",
4      requires="FALSE"
5  )
```

However, as before, this works in most cases but is slightly unsatisfactory, as it prevents this
operation on *all* maps, not just the map which is returned from the `referenceData` method.

The problem can be fixed by tracking the keys that are put into a map (with the MapKey(Object,
Map) relationship) and then placing a constraint on the end of the `referenceData` method that the
`Map` being returned does not contain the form command as a key. The specifications in Listing 6.14
do exactly this and allow the pragmatic analysis to find all of the erroneous plugins.

---

[6]As seen in Figure 6.1, `BaseCommandController` is a superclass of `SimpleFormController` that handles mapping a
form backing object to a command for the view to use.

Listing 6.14: Specifications to precisely describe correct usage of the `Map` in `referenceData`.

```
1   @Constraint(
2       op="Map.put(Object key, Object value) : void",
3       effect={"MapKey(target, key)"}
4   )
5
6   @Constraint(
7       op="EOM: AbstractFormController.referenceData(..) : Map",
8       trigger="MapKey(result, str) AND FormCommand(clss, str, target)",
9       requires="FALSE"
10  )
```

These specifications aren't without problems. While they are correct with regard to allowing and disallowing the right sets of plugins, the error produced is not in as useful of a location for the plugin developer. While the first attempt gave an error at the line where the command name was put into the `Map`, the second set delays the error until the return statement.

Both of these constraints show the tradeoff between creating a generic constraint that applies to all objects and creating more specifications which are specific to the problem. Which is "better" is dependent on several external factors, including the problem itself, the expected ways that a plugin developer might break the constraint, and the time of the framework developer. Since the Fusion language works with an abstract representation that is not directly tied to the heap, it provides framework developers with the flexibility to choose their own level of abstraction based upon their needs. In fact, the anticipated use of Fusion would be as a fire-fighting tool, where specifications are only written or refined on an as needed-basis. When a developer discovers a commonly broken constraint, she can create a small specification that will check most instances, and if it becomes a further problem, she can refine it later.

6.5 Generalizable properties of Fusion

In this thesis, I've shown Fusion to be able to specify the collaboration constraints found within the ASP.NET and Spring frameworks. While it is not possible to generalize from this to all frameworks, the Spring case study did give a sense as to what parts of this system might generalize easily to other frameworks, and what parts might not.

1. *Relationships generalize.* The relationship abstraction generalized well and did not change throughout the case study. Its flexibility allowed it to be used to specify not only pure Java examples, but also pure XML examples and mixed examples. The relationships themselves can even cross the boundaries of frameworks; as seen above, we created the MapKey(Object, Map) and ReadOnly(ServletRequest) relationships that are used by the Spring framework, but they are really owned and created by the Collections framework and Servlet framework respectively.

2. *Constraints generalize.* The form of writing the constraints with distinct predicates for the trigger, requirement, restriction, and effect also generalizes well. While there are several

kinds of specifications in Fusion that are specific to common paradigms (like the callback specification and the effect specifications), and we might make others to address common paradigms of other frameworks, all of them can be rewritten into the general constraint form.

3. *Operators do not generalize.* When I started this research, the only operator allowed in the "op" part of a constraint was a method call. This has expanded to cover constructors, beginning of method tags, end of method tags, and even an operator for checking a constraint only after the declarative files are processed. These were sufficient to cover the interaction paradigms that Spring has with its plugins, but such paradigms might be different for other systems. As seen in Table 6.2 field read and writes were also important for Spring, and one could imagine scenarios where even locking on a particular object is part of a collaboration constraint.

4. *Languages do not generalize.* Even for very similar languages, such as Java and C# or XML and ASPX, the language features that are used the most for framework interactions are the ones that are the most complex and the most distinctive to the specific language. To make this system truly work for C#, I would need to add support for properties, delegates, and partial classes, all of which play key roles in the ASP.NET framework. To completely work for Spring, Table 6.2 showed that Fusion needed to support JSP, OGNL, reflection, and even filesystem resources. While the declarative languages XML, ASPX, and JSP all have similar syntax, their form is distinct enough that each would require their own language for retrieving relationships.

While the Fusion language itself might not generalize beyond the common paradigms of Java and XML-based frameworks, it seems reasonable that the abstractions that Fusion is based upon, particularly the relationship and constraint abstractions, would generalize to other languages and paradigms.

Chapter 7

Adoptability

In my thesis, I have set out to create an *adoptable* specification and analysis tool to describe collaboration constraints and statically detect violations of them. In previous chapters, I have shown the functionality and scope of the system, but I did not discuss whether it was adoptable. That is, is the Fusion tool reasonable to use in practice?

While the best way to answer such a question would be to deploy the tool to a wide variety of industry projects, this is not feasible for an alpha-stage research project. Therefore, I have used the research literature to create a list of properties that an adoptable specification and analysis system must have. This list is by no means complete; it leaves out many properties such as a good user interface and integration with existing tools. However, I can show that Fusion does have several properties which are necessary, if not sufficient, for industrial adoption.

In this chapter, I will present a second case study done with Pradel, Aldrich, and Gross [86]. In this case study, we combined Pradel and Gross's specification miner [85] and Fusion to analyze the DaCapo benchmarks, a well-studied suite of program analysis benchmarks [18], to check collaboration constraints from the Java Standard Libraries. This case study shows several properties of Fusion that are necessary for adoption and are part of existing industry tools.

7.1 Reducing specification burden

One of the most important properties of an adoptable specification language is to reduce the cost of writing specifications as much as possible. Many commercial tools go as far as having no specifications at all, including Klocwork [64], Fortify [41], Findbugs [37], and Coverity [28]. Other tools, like JSure [70] and Spec# [88], reduce the specification burden by making languages that are highly modular so that the developers can specify as little or as much of the system as they like, thus allowing them to make their own cost-benefit tradeoff. Fusion also works on this model by allowing developers to specify as each constraint independently without specifying the entire framework.

Even writing these few specifications can be costly, as developers must learn a new specification language. To further reduce the specification burden, some tools have begun inferring

specifications through analysis. Inference is well-known within the type systems community, and entire languages, such as ML, are built with type inference in mind. Even popular industry languages, such as C#, have incorporated local type inference to reduce the burden of writing down type specifications [75].

While static inference can reduce type specifications, dynamic inference has been shown to capture more complex specifications. Both the Daikon research tool [48] and the commercial tool TestOne [6] use dynamic analysis to infer the pre- and post-conditions of methods. More recently, dynamic analysis has been used to infer multi-object protocols [67, 69, 85] similar to those described by Fusion. In our recent study we utilized these dynamically inferred protocols as specifications of collaboration constraints and used them to check programs without *any* developer intervention.

Our combined system and evaluation of it is written up fully in [86], but I provide a brief high-level description here. We ran the specification miner described in [85] on several samples runs of production-quality code. From these runs, the specification miner produces state machines based upon the calls it sees; a sample state machine from the `Iterator` protocol is shown in Figure 7.1. We translated each of these protocols into a Fusion specification. The actual translation is written up in [86] and is not necessary for this discussion. However, it is important to know that in order to retain precision, we created what we termed the "triple bookkeeping" system: we effectively translated the state machine in three ways. For each state machine, we created relationships from the states themselves, the operations used to transition, and the associations between each pair of objects in the protocol. This allows the analysis to regain precision from the other two sets of relationships even when one set losses precision.

The triple bookkeeping of the state machine creates some very complex constraint specifications. The protocol of Figure 7.1 is translated into 13 constraints, as shown in Listing 7.1, which utilizes 8 relationships. By contrast, the same protocol, specified by hand, only uses 3 constraints and 2 relationships, as shown in Listing 7.2. When specified by hand, the developer can take advantage of their global abstractions of the protocol, rather than doing more local transformations. In fact, Listing 7.2 is not only more concise, but also more precise as the protocol miner in [85] does not take advantage of the return values from methods (like `Iterator.hasNext()`). While the inferred constraints are far more complex, they took no intervention from the developer beyond running the specification miner on sample programs.

7.2   Scalability and Performance

Scalability is another important property of an adoptable program analysis. In [14], the Coverity team explains that in order for their tool to be marketable to companies, they have to be able to run their analysis tool in an overnight build of 12 hours. Based on their experience, an analysis tool needs to process 1400 LOC a minute, which comes to about 1 MLOC an hour. In extreme cases, such as where they are running on over 10 MLOC, they can get away with a 24 hour analysis time.

To truly evaluate scalability, I would need to show the run times for samples of different sizes of programs with different numbers of specifications; I will not be doing that here. Instead, I will demonstrate that Fusion can achieve the high bar set by Coverity with regards to performance and I will identify the aspects that lead to scalability and performance concerns within Fusion.

Listing 7.1: Automatically generated specifications for the state machine shown in Figure 7.1. While these specifications appear to be unnecessarily repetitive, the repetition is necessary for more complex inferred protocols.

```
1  Constraint(op ="Iterator.remove() : void",
2     trg ="(fsm162(target))",
3     req = "TRUE AND remove(target)")
4  Constraint(op ="Iterator.remove() : void",
5     trg ="(fsm162(target)) AND (s1(target))",
6     eff = {"hasNext(target)", "!s1(target)", "s0(target)", "!next(target)", "fsm162(target)",
7           "!s3(target)", "!remove(target)"})
8  Constraint(op ="Iterator.remove() : void",
9     trg ="(fsm162(target))",
10    eff = {"hasNext(target)", "!s1(target)", "s0(target)", "!next(target)", "fsm162(target)",
11          "!s3(target)", "!remove(target)"})
12 Constraint(op ="Iterator.next() : Object",
13    trg ="(fsm162(target))",
14    req = "TRUE AND next(target)")
15 Constraint(op ="Iterator.next() : Object",
16    eff = {"hasNext(target)", "remove(target)", "!next(target)", "fsm162(target)", "!s3(target)",
17          "s1(target)", "!s0(target)"})
18 Constraint(op ="Iterator.next() : Object",
19    trg ="(fsm162(target)) AND (s3(target))",
20    eff = {"hasNext(target)", "remove(target)", "!next(target)", "fsm162(target)", "!s3(target)",
21          "s1(target)", "!s0(target)"})
22 Constraint(
23    op ="Iterator.next() : Object",
24    trg ="(fsm162(target))",
25    eff = {"hasNext(target)", "remove(target)", "!next(target)", "fsm162(target)", "!s3(target)",
26          "s1(target)", "!s0(target)"})
27 Constraint(op ="Iterator.hasNext() : boolean",
28    trg ="(fsm162(target))",
29    req = "TRUE AND hasNext(target)")
30 Constraint(op ="Iterator.hasNext() : boolean",
31    trg ="(s0(target)) AND (fsm162(target))",
32    eff = {"hasNext(target)", "!s1(target)", "next(target)", "s3(target)", "fsm162(target)",
33          "!remove(target)", "!s0(target)"})
34 Constraint(op ="Iterator.hasNext() : boolean",
35    trg ="(fsm162(target)) AND (s1(target))",
36    eff = {"hasNext(target)", "!s1(target)", "next(target)", "s3(target)", "fsm162(target)",
37          "!remove(target)", "!s0(target)"})
38 Constraint(op ="Iterator.hasNext() : boolean",
39    eff = {"hasNext(target)", "!s1(target)", "next(target)", "s3(target)", "fsm162(target)",
40          "!remove(target)", "!s0(target)"})
41 Constraint(op ="Iterator.hasNext() : boolean",
42    trg ="(fsm162(target)) AND (s3(target))",
43    eff = {"hasNext(target)", "!s1(target)", "next(target)", "s3(target)", "fsm162(target)",
44          "!remove(target)", "!s0(target)"})
45 Constraint(op ="Iterator.hasNext() : boolean",
46    trg ="(fsm162(target))",
47    eff = {"hasNext(target)", "!s1(target)", "next(target)", "s3(target)", "fsm162(target)",
48          "!remove(target)", "!s0(target)"})
```

Figure 7.1: An inferred state machine on the Iterator protocol.

Listing 7.2: Manually written specifications for the state machine shown in Figure 7.1.

```
1   Constraint(
2       op ="Iterator.hasNext() : boolean",
3       eff = {"?HasNext(target) : result"})
4   Constraint(
5       op ="Iterator.next() : Object",
6       req ="HasNext(target)",
7       eff = {"!HasNext(target)", "Removable(target)",})
8   Constraint(
9       op ="Iterator.remove() : void",
10      trg ="Removable(target)",
11      eff = {"!Removable(target)"})
```

Once we had the 223 inferred constraint specifications as described above, we ran Fusion with the specifications on the entire DaCapo benchmark. The DaCapo benchmark is a 1.5 MLOC benchmark of production code used for program analysis [18] and provides a useful measure for how well our system works. While primarily used by dynamic analysis tools, it has recently been used by static tools as well, including some related work [20, 25, 45, 78]. The size of each program within the benchmark is shown in Table 7.1. Fusion successfully ran overnight on this benchmark on an Intel machine with a 3.0 GHz quad-core processor and 8GB of RAM. While not running at speeds of 1MLOC per hour, we made few optimizations and it ran overnight easily.

Simply running an intra-procedural analysis on 1.5 MLOC is fairly trivial though. In practice, I found that three aspects beyond the lines of code significantly contributed to the performance of Fusion: the number and complexity of these specifications, the number of times the specified API was used, and the number of options produced by the points-to analysis. The complexity of the specifications affect performance because there are simply more relationship effects to make and more relationships to keep track of. The frequency that the API was used affected how often an instruction in the program matched an operation in the specifications; in our case study, this

Program	Description	LOC
avrora	Analysis of microcontrollers	69,393
batik	SVG toolkit	186,460
daytrader	Application server benchmark	12,325
eclipse	Software development platform	289,641
fop	Output-independent print formatter	102,909
h2	SQL relational database	120,821
jython	Python interpreter	245,016
lucene	Text indexing tool	124,105
pmd	Source code analyzer	60,062
sunflow	Photo-realistic rendering system	21,970
tomcat	Servlet container	161,131
xalan	XML processing	172,300
Sum		1,566,133

Table 7.1: DaCapo programs used for the evaluation of the inferred specifications. Table from [86].

happened 606,706 times. Not all of these resulted in an error, or even a triggered constraint, but each match takes time because we have to check the constraint to see if it is triggered.

The final piece, the points-to analysis, was a surprisingly large factor for scalability and performance. In most cases when a constraint was triggered, there would be only a few substitutions σ produced by the points-to analysis, as described in Chapter 5. However, certain methods would produce thousands of substitutions; this frequently occurred in methods with many string concatenations. String concatenations produce a temporary string as a result, so it was not unusual for a single method to have 15-20 potential labels for `String`s. As any of these could be aliased, there is a huge explosion in the number of possible substitutions. Code with string concatenations would not be a problem normally, but we had several protocols about the `StringBuffer` API, so these constraints matched instructions frequently. The sheer number of substitutions took surprisingly long to check, and in tests, a single method like this would take hours and seem to hang the analysis. To prevent this from occurring, we stopped analyzing a method if a constraint ever matched with over 100 substitutions or if it takes longer than 30 seconds of analysis time. In practice, this occurs in less than 1% of methods, so we determined this to be a good tradeoff between precision and performance. Coverity uses similar techniques in order to keep the analysis time within an overnight run[14].

7.3 Precision

For a static analysis tool to be adopted by industry, its results must be precise enough to be cost effective. Both false negatives and false positives decrease the value of a tool, and successful industry tools provide a good balance between these. Each false negative from a tool decreases its potential value, and the total cost of using the tool, including purchase cost and setup costs, must be correspondingly lower. False positives also decrease value, though in a very different

way. Each false positive costs (expensive) developer time to investigate. Worse yet, if there are many false positives, developers will be unable to find the true positives and will stop using the tool altogether. For this reason, sound analyses have had little little headway in industrial use. Even unsound analyses tools must be mindful of this; in [14], the Coverity team explained their experiences with false positives:

> *False positives do matter. In our experience, more than 30% easily causes problems. People ignore the tool. True bugs get lost in the false. A vicious cycle starts where low trust causes complex bugs to be labeled false positives, leading to yet lower trust....We aim for below 20% for "stable" checkers. When forced to choose between more bugs or fewer false positives we typically choose the later.*

In Chapters 4 and 5, the pragmatic variant worked very well; in fact, it was perfectly precise. However, this was on very limited examples. Each example program was relatively small and was generated from snippets of code from internet help forums. In earlier chapters, I noticed that, in addition to the variant, there were two other factors that impact precision: the precision of the points-to analysis and the precision of the specifications. While neither was a serious issue in the Spring case study, the DaCapo case study thoroughly tested both of these factors.

The DaCapo benchmarks are entirely large, open source programs that are currently in production. As we expect to see relatively few bugs in previously-tested production code, we will expect our false positive rate to be high. Additionally, this code has much more complex aliasing patterns, and without any aliasing control specifications, like fractional permissions [21] or ownership types [27], it is going to be very difficult for a points-to analysis to produce precise results. Therefore, we must expect Fusion to perform worse accordingly.

The specifications used in this case study are also not very precise. As the 223 protocols are dynamically inferred, they can only capture the parts of the protocol that the training runs actually used. To make matters worse, the translation from these protocols into specifications are not as precise as human specifications, and the inferred protocols do not capture value-based information, like whether the return value from `hasNext` is `true` or `false`. To remove the worst offenders, we employed an automatic filtering system, described in [86] to prune out any protocols with signs of being an imprecise protocol. For example, one pruning mechanism was to remove protocols that were not seen at least a certain number of times in the training programs. Pruning out protocols removed large numbers of warnings; the complete analysis reported 993 warnings before pruning, but only 81 after pruning.

Even with complex aliasing patterns and imprecise specs, the analysis performed reasonably. While the pragmatic analysis did not fare well, the complete analysis had a false positive rate of 51% and found 41 real issues in the DaCapo program, including 26 defects and 15 code smells. Table 7.2 shows a breakdown of the results. Most of the false positives were from incomplete protocols, that is, imprecise specifications. There were only two false positives from imprecisions in the points-to analysis.[1] Overall, while it does not achieve the 30% marker given by Coverity, the analysis performed well in a very difficult environment and might do considerably better in other environments.

---

[1]Given that these results are for the complete variant, which uses the must-like analysis, there are probably many false negatives from this. The only way to evaluate how many would be to analyze the results of the sound variant to find them all.

Kind of Issue	Number
Total	81
False Positive	40
Incomplete Protocol	30
Imprecise aliasing	2
Extended, specialized protocols	8
True Positive	41
Bug	26
Code smell	15

Table 7.2: Results from running inferred specifications on the DaCapo programs using the complete analysis and after automatically pruning bad protocols. In addition to incomplete protocols and imprecise aliasing, eight false positives were from programs that extended existing protocols with their own specialized semantics.

Listing 7.3: Bug found by the iterator specifications in Listing 7.1.

```
1  Map comparators = ...
2  Iterator i = comparators.values().iterator();
3  for (Comparator c = (Comparator) i.next(); c != null; c = (Comparator) i.next()) {
4      ...
5  }
```

Most of the defects found were from only a few very commonly used protocols. Listing 7.3 gives an example of a defect found on the `Iterator` protocol; this was found using the constraint specifications from Listing 7.1. In this listing, the code assumes that a call to `next` will return null if there is no next operator, which is incorrect according to the specification of Iterator [103]. The analysis also found several issues that we classified as a code smell. These issues were a fault in the code that would not cause an error, but make the code less readable. Listing 7.4 shows code that closes a stream twice; while not technically an error, this is unnecessary.

7.4 Usable error reports

The final property to discuss is the ability for the analysis to produce understandable error messages. In the article on their experiences at Coverity, the team mentioned the need for understandable error messages several times and seemed to think this was their biggest technical hurdle:

> *Further, explaining errors is often more difficult than finding them. A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive.*

That is, even a tool that produces very few false positives may have a high false positive rate in practice if the error messages themselves are not understandable. This has become so important to them that they "have completely abandoned some analyses that might generate difficult-to-understand reports" [14]. The problem is not uncommon; the FindBugs team has a website that

Listing 7.4: Code smell found by inferred specifications

```
1   BufferedReader in = null;
2   try {
3       in = new BufferedReader(...);
4       ...
5       in.close();
6   }
7   finally {
8       if (in != null) {
9           try {in.close();}
10          catch (IOException e) { ... }
11      }
12  }
```

describes every defect, with examples for some, so that people will not mistakenly mark warnings as false positives [38]. In all the industry tools I have used, the error messages are pre-defined and it is easy to access examples and further discussion of the error. This is practical for most tools to do as the checkers are all provided by the tool company; end-users never or rarely write their own specifications and never use specification languages as complex as Fusion.

While I could depend on the framework developer to write their own error messages for each constraint specification, it seems unlikely that developers would do so and more likely that this would just be a hinderance to adoption. On the other hand, just showing the failing constraint to the user as a logical predicate is insufficient for explaining the error. This would require the plugin developer, who already is unsure of the problem (after all, they wrote the code) to understand a new specification language and understand the abstractions that the framework developer chose to use.

As a step toward fixing this situation, I created error reporting logic (ERL) to automatically generate human-readable error messages from failing first-order logic propositions [26]. The premise of ERL is to find the sub-parts of the proposition which contribute to the failure and must be fixed. ERL breaks apart these contributing pieces so that each error message represents a single action that a developer must take to resolve the error. Therefore, a failing conjunction where both sides are failing results in two error messages, as there are two distinct tasks. On the other hand, a failing disjunction where both sides are failing results in a single error messages that allows the user to select between two tasks. A conjunction with only one side failing will only show one error message, as the system will only show the sub-parts that need to be changed rather than the entire failing proposition.

In [26], we evaluated ERL on AcmeStudio which, like Fusion, uses first-order predicate logic specifications that may not have a human-readable error message [5]. Our qualitative analysis suggested that the more focused error messages helped developers to find and fix their errors. ERL is currently being added to Fusion, and I expect the benefits to Fusion will be similar to the benefits found with AcmeStudio. While this is still not as good as a detailed English description with examples, this is a major improvement that could be added to other logical specification systems as well, including [16, 66].

7.5 Future work for adoptability

In addition to improving further on the above properties, there are several other steps needed to truly make Fusion adoptable by industry.

1. *Visualizations* of the relationships and the aliasing patterns at each line of code would make it much easier to determine whether a warning was a true positive or false positive, or even whether the specification itself is incorrect. While we do not have such a visualization now, we do have a textual output that shows the lattices at a highlighted line, and I have found this to be extremely helpful when trying to understand the cause of the error in complex code from DaCapo.

2. *Adjustments* to inferred constraints, done by the plugin developer on the fly, would make inferred protocols much more tractable. While dynamic inference creates mostly correct protocols, there were several cases where the protocol was just slightly off and causing false positives. The ability for the plugin developer to change this on the fly, perhaps through a visualization or perhaps automatically by marking false positives, would greatly improve the results.

3. *Suggestions* to fix the errors would improve the error messages. Even with ERL, the error messages reference relationships, which are a framework developer's abstraction of their API. It would be much better if the plugin developer received suggestions for how to fix the problem in terms of their own code, rather than in terms of a foreign abstraction.

4. *Support for file resources* would greatly increase the scope of defects Fusion can find. The Coverity team has a law: "You can't check what you can't see". [14] Right now, many important files are effectively invisible to Fusion, and most other analysis tools, because they are accessed through dynamically created filepaths that a static analysis tool can't yet follow. This will enable many other kinds of checking, including checking JSP files for compatibility with associated Java and XML files in Spring.

Chapter 8

Related Work

There are four primary areas of related work. First, there are other approaches to helping plugin developers understand frameworks, some of which take a tutorial-based approach, and some of which take a predicate-based specification approach; both are described here. There are also other uses of relationships as specifications. The analysis itself is very similar to many existing shape analyses and can even be encoded within some well-known analysis frameworks. Finally, there are three similar areas of work (typestates, tracematches, and session types) which also seek to encode protocols, including a few systems that support multi-object protocols.

8.1 Tutorial-based framework assistance

Most of the work on improving the usability of software frameworks has been through either documentation of the framework design or through tutorial assistance. Johnson's early work on software frameworks described them as compositions of design patterns [59, 60]. This was followed by research that aimed to formalize and extract these design patterns [30, 44, 97]. However, design patterns alone have been insufficient for specifying frameworks. While they provide information at a high level of abstraction, they become unwieldy when used to describe lower-level constraints as seen here because they cannot handle all the points of variation without an explosive number of patterns to express them.

More recent work on frameworks helps developers by documenting tutorial-like use cases [36, 43, 71, 89]. These use cases are more flexible than the original pattern-based work as they do not attempt to describe frameworks using external patterns; rather, they work within the abstractions of the framework. This allows them to describe the specific steps that the plugin developer must take to achieve some task. While this work can help a plugin developer find the right API and get started using it, it does not help a plugin developer expand beyond the tutorial. This body of work is complementary to the work in this thesis; the tutorial-style helps a developer get started on a good path, and Fusion ensures that as developers expand their application that they stay away from bad paths.

8.2   Formal specifications of frameworks

SCL [55] allows framework developers to create a specification for the structural constraints for using the framework. Unlike Fusion, it does not handle the semantic aspects of the protocol, including object identity or values.

Like Fusion, Contracts [52] also view associations between objects as a key factor in specifying systems. A contract also declares the objects involved in the contract, an invariant, and a lifetime where the invariant is guaranteed to hold. Contracts allow all the power of first-order predicate logic and can express very complex invariants. Contracts differ from Fusion because they do not check the conformance of plugins and the specifications are more complex to write due to their higher level of expressive power.

Others have noted the importance of handling inheritance for code reuse purposes. Dhara and Leavens noted the problem in [33] and relaxed the constraints in JML to better handle this problem. Parkinson and Bierman introduced a verification technique based on separation logic that handle subclasses which break behavioral subtyping [82]. Parkinson and Bierman's approach is particularly interesting because they were able to handle broken behavioral subtyping and did so in a modular analysis. Fusion does not do this and assumes global knowledge of constraints; however, Fusion must have global knowledge anyway in order to handle constraints which are not class invariants.

Relationships are not a new construct to specification languages. Bierman and Wren formalized UML relationships as a first-class language construct [17]. The language extension they created gives relationships attributes and inheritance, and developers use the relationships by explicitly adding and removing them. Balzer et. al. expanded on this work by describing invariants on relations using discrete mathematics and support semantic invariants and invariants between several relations [10]. In contrast to previous work, the relationships presented in this paper are added and removed implicitly through use of framework operations, and if inferred relationships are used, they may be entirely hidden from the developer.

This work also has some overlap with formal methods, particularly in describing the relationships and invariants of code [40, 66]. These formal methods verify that the specified code is correct with respect to the specification; this is also called "implementation-side verification". Instead, we are checking the unspecified plugin code against the framework's specification; this is known as "client-side verification". Other formal methods [57, 98] focus on a detailed description of the entire system. These systems also allow developers to model the invariants between objects. However, the checkers for these systems are meant to stand on their own, without any ties to executable code. The closest work in formal methods is [8], as it also allows for framework developers to define their own constraints. All of these checkers expect to verify invariants of the system that are true throughout the lifetime of the application. Instead, Fusion checks constraints that only hold true for specific contexts, and it takes into account that the relationships between objects might change over time.

Many verification and typechecking systems [4, 19, 24, 39, 72] have proposed doing a static analysis to verify as much of the system as possible, and then using a dynamic analysis for unverifiable program points. Fusion could be easily modified to also take this approach; any issue found by the sound variant, but not by the complete variant, would require instrumentation for a

Table 8.1: Comparison of closely related work. These four areas are likely isomophic solutions with different design choices in the solution space. The cited works are only those which handle multiple objects in some way; they are many more papers in each of these areas.

	Specified a valid protocol	Specifies erroneous paths of the protocol
State-based	Typestate [16, 65, 79]	Relationship constraints [58]
Operation-based	Session Types [54]	Tracematches [78]

runtime check.

8.3 Logical analyses

The Fusion analysis is similar to a shape analysis, with the closest being TVLA [92]. TVLA allows developers to extend shape analysis using custom predicates that relate different objects. Fusion constraints could be written as custom TVLA predicates, but the lower level of abstraction would result in a more complex specification and would require greater expertise from the specifier.

While the mechanism to infer relationships is clearly a Prolog engine, the main analysis can also be modeled as a logic program. In fact, I did model the `DropDownList` example constraint in Datalog, in hopes of feeding it into BDDBDDB and taking advantage of the pointer analysis described in [114]. I found it to be troublesome to model data-flow as it is not built-in and must be modeled at a low level. Additionally, I needed higher-order functions to make the technique practical for framework developers to write the specifications, and Datalog does not currently support this.

8.4 Typestates, Tracematches, and Session types

The most related work to Fusion are typestates, tracematches, and session types, all of which seek to describe object protocols. None of the work described here can handle declarative artifacts, though a few can specify semantic aspects of constraints, extrinsic constraints, and/or multi-object constraints, with some limitations. Table 8.1 shows how these four areas are related and the different properties of each. I will first describe how each research area is related to relationship constraints, and I will come back to the comparison in Table 8.1 at the end of the chapter.

Typestates [32] provide a mechanism for specifying a protocol on a single object by using a state machine. There have been several approaches to inter-object typestate. Kuncak et al. manipulated the typestate of many objects together through their participation in data structures [65]. Nanda et al. take this a step further by allowing external objects to affect a particular object's state, but unlike relationships, it requires that the objects reference each other through a pre-defined path [79]. Bierhoff and Aldrich add permissions to typestates and allows objects to capture the permission of another object, thus binding the objects as needed for the protocol [16]. Relationships can combine multiple objects into a single state-like construct and is more general for this purpose than typestate; it can describe all of the examples used in multiple object typestate work.

However, Fusion does not contain a built-in aliasing system, and therefore it may be less precise if there is significant aliasing.

With respect to the specifications, relationships are more incremental than typestate because the entire protocol does not need to be specified in order to specify a single constraint. Additionally, the plugin developer does not add any specifications, which she must do with some of the typestate approaches. However, typestate analyses aim to be sound, and can also check that both the plugin and the framework meet the specification. The relationship analysis assumes that the framework properly meets the specification and only analyzes the plugin.

Tracematches have also been used to enforce protocols [112]. Unlike typestate, which specifies the correct protocol, tracematches specify a temporal sequence of events which lead to an error state. This is actually more similar to how Fusion specifies constraints. In tracematches, this is done by defining a state machine for the protocol and then specifying the bad paths.

The tracematch specification approach is similar to that of relationships; the main difference is in how the techniques specify the path leading up to the error state. Tracematches must specify the entire good path leading up to the error state, which leads to many specifications to define a single bad error state. In cases where multiple execution traces lead to the same error, such as the many ways to find an item in a `DropDownList` and select it incorrectly, a tracematch would have to specify *each* possibility, as seen in Listing 8.1. Instead, Fusion allows us to specify a relationship predicate that triggers the check, and we separately write specifications on the good paths leading up to the check to produce the relationships necessary for the trigger. This difference affects how robust a specification is in the face of API changes. If the framework developer adds a new way to access `ListItems` in a `ListControl`, possibly through several methods calls, the existing tracematches will not cover that new sub-path. However, all the constraint specifications in Fusion will continue to work if the sub-path eventually results in the same relationships as other sub-paths.

Unlike relationships, tracematches are enforced both dynamically and statically using a global analysis [19]. The static analysis soundly determines possible violations, and it instruments the code to check them dynamically. Bodden et al. provide a static analysis which optimizes the dynamic analysis by verifying more errors statically [20], and Naeem and Lhoták specifically optimize with regard to tracematches that involve multiple objects [78] . While this work handles multiple objects and object identity, it cannot currently handle value-based constraints. In particular, tracematches can be used to determine that a call to `hasNext` appeared before a call to `next`, but cannot check whether the call returned `true`.

As seen, typestate and tracematches are state-machine based approaches, but this approach generally breaks down in the presence of multiple objects. The core of the problem is that all objects much be accessible to start up the state machine, and in many of the multiple-object constraints, only a couple objects exist at a time. The typestate approach given by Bierhoff [15] attacks this issue by using a permission capture to hold onto the object permissions for later use in the protocol, while the tracematch approach must specify all possible paths up to the point where the first object was bound [77]. Fusion avoids this by abstracting away the earlier binding of objects into relationships and then composing relationships together into logical predicates.

Session types [53] were originally created to describe the protocol between two processes. They were later extended to allow for multi-party sessions [54]. Like typestate, session types describe the protocol to follow, instead of the bad paths. However, like Fusion and tracematches, session

Listing 8.1: The tracematch to specify the `DropDownList` selection protocol from Vignette 3.1.

```
1   tracematch(DropDownList ddl, ListItemCollection coll, ListItem newSel, ListItem oldSel) {
2     sym getCurrent after returning(oldSel):
3       call(* DropDownList+.getSelectedItem()) && target(ddl)
4     sym deselect after:
5       call(* ListItem+.setSelected(boolean select)) && target(oldSel) && select == false
6     sym getList after returning(coll):
7       call(* DropDownList+.getItems()) && target(ddl)
8     sym getItem after returning(newSel):
9       (call(* ListItemCollection+.findByValue(...)) ||
10      call(* ListItemCollection+.findByName(...))) && target(coll)
11    sym select after:
12      call(* ListItem+.setSelected(boolean select)) && target(newSel) && select == true
13
14    getList getItem select (getCurrent deslect+)+ |
15    getCurrent getList getItem select deselect+ |
16    getList getCurrent getItem select deselect+ |
17    getList getItem getCurrent select deselect+
18    {
19      throw new RuntimeException("Need to deselect the existing object before selecting");
20    }
21  }
```

types describe the specification globally; this allows them to easily handle extrinsic constraints. After the protocol is specified as a session, each participant is verified against the protocol.

It's important to note that the "party" abstraction used in multi-party session types does not entirely map to objects in a multi-object protocol. A party is a process, or perhaps, a component. Therefore, in a situation where a plugin interacts with the framework through four objects, as in the `DropDownList` problem from Vignette 3.1, there are only two parties: the framework, and the plugin. However, it seems this is an arbitrary division; we could just as easily divide the framework into its component parts and call this a five-party protocol (the 4 objects, plus the plugin that is calling them).

As described, type systems, trace matches, and state machines are all related to relationship constraints and to each other. Table 8.1 shows two axes where these areas have fundamentally different design choices to specify the same kinds of protocols. While I and others hypothesize that these areas are isomorphic, the design choices affect the ease of specifying different types of protocols.

The first axes is in specifying the valid parts of the protocol or the erroneous parts. Both typestate and session types specify the correct way for objects to interact, and any deviation from the specified protocol is an error. On the other hand, tracematches and relationship constraints specify the bad usages that can cause an error, and all usages otherwise are deemed acceptable. The choice of which is "better" is dependent on whether the protocol in question has more good paths or more error paths.

The second axes is on how the protocol is specified. Typestates and relationship constraints use a state-based approach where the specifications are on a state-like abstraction. On the other hand,

tracematches and session types use operation-based specifications; they specify the path of interest in a regex-like syntax on the operations. Again, which choice is "better" is dependent on the protocols. If we expect protocols where operations and states have a near 1:1 relationship (like a `File` protocol), an operation-based approach is a clean abstraction. However, if several operations transition to the same state, or if a single operation transitions to different states depending on the current state, a state-based abstraction is cleaner, as seen with the example from Listing 8.1.

The Fusion system is unique from the related specification and verification systems in several ways. First, it completes the design space in Table 8.1 by providing a state-based specification for erroneous protocols. Second, it is the first system shown to be able to specify and analyze constraints that span both code files and declarative artifacts. Finally, it is the only system that provides not just a sound analysis, but also a complete variant and a pragmatic variant in order to provide more cost-effective results.

Chapter 9

Conclusion

In this dissertation, I made the following thesis statement:

> *Collaboration constraints are inherent to the design of software frameworks but are burdensome for plugin developers. These constraints can be defined by specifications that describe the relationships between objects and how relationships change, and an adoptable static analysis can check that code conforms to the specified constraints.*

This thesis presents both a new problem, previously undiscussed in the research literature, and a solution that, while built upon a foundation of prior work, also adds its own contributions to the community.

9.1 Frameworks and Collaboration Constraints

I have argued that collaboration constraints are the outcome of several interacting quality attributes of software frameworks, and that they cannot be removed from reusable software components, such as frameworks, without sacrificing other desirable quality attributes. Chapter 2 took an architectural view of software frameworks and showed that the designs of reusable components, including software frameworks, must address a tricky tradeoff. These designs need three quality attributes to be reusable: usability of the API of the component, utility of the component, and versatility of clients to the component. However, Chapter 2 argues that these form an inherent tradeoff; only two can be maximized.

The choice in this tradeoff is determined by the business drivers of the component. Some, like many libraries, maintain a highly usable APIs by decreasing utility and versatility. Others, like software frameworks aim to increase utility with architectural reuse and to still maintain high levels of versatility; in these cases, the usability of the API suffers. This is the first work to present software framework designs as inherently less usable, yet for good reasons rather than for accidental ones. Indeed, framework design is a wicked problem [23].

Collaboration constraints are one result of trading away usability in reusable components. These abstract, state-based preconditions across several objects cause developers problems to the

111

point that they must post on help forums to understand why their code doesn't work when they violate a constraint. In Chapters 3 and 6, archival analyses of the ASP.NET and Spring frameworks help forums showed that these constraints had several common properties.

1. Collaboration constraints involve multiple types and objects.

2. Collaboration constraints are often extrinsic to a type.

3. Collaboration constraints involve semantic properties such as object identity, primitive values, state, and ordering of operations.

4. Collaboration constraints span many kinds of files and data, including declarative artifacts.

While they may occur less often in some types of programs, the archival analyses showed that these constraints are burdensome for developers. When plugin developers violated a collaboration constraint and posted their problem on the forum, they might wait for hours or days for a response. The responding developers frequently addressed the same collaboration constraint multiple times, but the old postings were not found by plugin developers, perhaps because the runtime form of the error was different depending on how the constraint was broken.

This is the first work to recognize collaboration constraints as a unique class of constraints and to show that they are common and troublesome for developers.

9.2   Relationships

To describe collaboration constraints and detect violations of them in plugin code, this dissertation introduces Fusion, a specification and static analysis system. While there are many possible methods to address this problem, static analysis meets my goals well; it allows developers to detect their errors earlier than run time, yet it can work with existing languages and frameworks. I specifically aimed for a system with modular, lightweight specifications to improve the adoption potential of the system.

Fusion uses *relationships* to describe collaboration constraints. As the relationships in Fusion are abstracted from program code, they can be used modularly; it is not necessary to specify an entire class to show what each relationship means in concrete semantics. This also means that relationships are an abstraction that can span across programming language boundaries.

This dissertation has shown that relationship-based specifications, as embodied by Fusion, can capture all four of the properties above which are common in collaboration constraints and that it is the first specification language of its kind to do so. Additionally, I have shown that it is able to specify many real collaboration constraints in the ASP.NET and Spring frameworks. Each example was specified modularly with few specifications. The examples also showed the variability of the specification language; a single constraint could be specified several ways depending on the desired goals of the framework developer.

The analysis itself is a straightforward dataflow analysis, but it also makes a unique contribution to static analysis. The analysis has three variants: a sound variant that can guarantee that passing programs are free of defects, a complete variant that can guarantee that all issues it finds are real violations, and a pragmatic variant that makes no guarantees but seeks to make

a cost-benefit tradeoff. This is the first analysis of its kind, and the case studies provided some interesting results about the tradeoff. While the pragmatic variant was clearly the best for small, untested programs of the type that are posted on forums, the complete variant provided better results for large, production-quality code using inferred specifications. This is not a preset configuration, rather, it is another point in the tradeoff space that might be changed depending on the use of the verification system. Such a knob broadens the possibility of using static analysis.

9.3   Future work

There are many potential avenues for future work, ranging from studies of socio-technical ecosystems to improvements in usability of verification systems to new programming languages.

In my studies of software frameworks, I found that some framework forums, like ASP.NET, were exceptionally active, while others, like Ruby-on-Rails, seemed dead by comparison. I noticed that the active frameworks had carefully cultivated their ecosystems and the surrounding technologies. For example, in ASP.NET, framework developers were very active on the forums, there was a ranking system which designated top members as "MVP"s, and there was a built-in means for marking responses as having solved the original problem. What is the effect of these features on the activity of the forum, and what is the effect to the entire ecosystem of the framework?

It would be interesting to find out what makes for successful uses of forums and find ways to encourage developers to use them in this way. In the study, it seemed that posters who got helpful responses posted more code than others, yet carefully crafted the smallest example that would reproduce their error. This of course takes time, but perhaps there are technical means to assist developers in creating these smallest reproducible examples.

This work highlighted the need for more attention to the usability of verification systems. While the work on error reporting logic was an improvement to error messages, these messages are still written in terms of the formal specification, rather than in terms that the plugin developer would understand. As the plugin developer is already having difficulty understanding the API, it seems unreasonable to require them to learn the formal specification of the API as well. Yet, all specification and verification systems seem to make the assumption that it is better to require developers to understand a formal specification. This would require developers to not only learn the formal language, but also to understand all the aspects of the specification, including those that they are not using. If a developer forgot to check `hasNext` before calling next, is it really necessary for them to understand the details of concurrent modification problems? Perhaps, however, we can improve on this and make suggestions to the developer on how to fix their program within terms of their own code. This would allow developers to quickly move through their current task, yet the specifications could still be available for exploring and understanding the API.

Finally, this work has shown the need for a programming language specific to the needs of configuration files, such as those seen in Eclipse, Spring, Hibernate, and others. These configuration files are frequently written in XML, which is intended as a data markup language. However, as seen in the case studies, these configuration files do more than act as a data repository; they create objects, assign objects to fields, and even handle control flow. Yet XML was not intended as

a programming language, and the technologies that support it, such as XPath and XQuery, are not sufficient for describing the deep semantics of these files.

While purists may suggest that these functions should be done in the programming language of the framework, this is not sufficient either. These frameworks specifically moved away from this model because the base programming languages had too many additional abstractions that made this difficult; the extensibility of XML makes it easy to use for configuration files. Additionally, it allows for the configuration file to by changed at run time, not at compile time. This means that the same codebase can be deployed to multiple environments without recompiling each time, and the configuration file can be changed dynamically with the environment. Further still, as such changes are normally handled by an IT professional rather than the programmer, XML is a common, easy-to-learn syntax that an IT professional can easily learn.

Experts in programming languages would make a different suggestion: these configuration files clearly represent a domain-specific language. Therefore, framework developers should create a new language, specific to their needs, for these configuration files. While this is possible, and while there are many good tools out there to help in this process, it still is not a satisfactory solution. The plugin developers would have to learn a new syntax just to learn the framework; XML works well because it is a known syntax, and while the semantics might change, there are some pieces which are consistent, such as containment through nesting nodes.

Instead of using XML or creating domain specific languages for each framework, I believe that the best solution would be a language for configuration that can be used by all frameworks. This would get the benefits of XML (a common language and shared syntax for all frameworks) yet also provide a set of language features that make sense for configuration. Possible language features might include objects, awareness of the filesystem, built-in string manipulation, and an extensible semantics. These are only potential ideas though, and there need to be further studies of configuration files before creating such a language.

9.4   Tradeoffs, tradeoffs, tradeoffs...

Tradeoffs have been a recurrent theme in this dissertation and have appeared in both anticipated and unanticipated ways.

There was an anticipated tradeoff in the static analysis. An analysis cannot find all and only true positives; there must be false results. By creating three variants of the analysis, I was able to explore the extremes of this tradeoff (soundness and completeness) and one point in the middle (pragmatic) to determine which was most useful in practice. The answer was dependent on the specifications used and the complexity of the analyzed code. The pragmatic variant as a clear winner for precise, handwritten specifications analyzed on simple, under-development code, but the complete variant was best for imprecise specifications analyzed on highly-complex, well-tested production code.

An unanticipated, though unsurprising, tradeoff came from the specifications themselves. As discussed in Chapter 6, there are many tradeoffs in the precision of the specifications, the complexity and cost of writing them, and the quality of the results. While it is not terribly surprising that a more precise specification is more complex and difficulty to write, what was surprising was that in some cases, like Section 6.4.4, the error given was more useful from the less precise speci-

fication. Even though the less precise specification might give a false positive, such instances are rare enough in this case that we would trade that for increased quality of the true positives. The flexibility of the specification language allowed me to describe each of the example problems in several ways and select the most beneficial. Alternatively, Chapter 7 showed how fully-automated techniques could generate specifications; while such specifications are comparatively very imprecise, they take relatively little cost to create. One can even imagine a semi-automated tool that would provide further points on this tradeoff space.

The final tradeoff in this dissertation is not in the solution space, but in the problem domain itself. As described in Chapter 2, reusable component design is fraught with complex tradeoffs. It is possible to eliminate collaboration constraints and all the problems they produce, but only at the expense of other quality attributes of functionality. Each reusable component comes with a unique set of business drivers, and so while there is design guidance available for how to manage this tradeoff, there is no solution for how to actually solve it. A designer must use their own judgment to select the most ideal location in this tradeoff space and attempt to limit the resulting damage from collaboration constraints as much as possible.

This dissertation does not present a single, one-size-fits-all solution because there is not a singular problem. Collaboration constraints exist in many different settings, and there are a variety of situations for both specification and analysis. A truly adoptable verification system allows itself to be customized easily for each new situation it might encounter, thus increasing its versatility. This dissertation has shown several ways this can occur, many of which can be used by other verification systems. Through this variability, perhaps we can overcome the inherent usability problems of software frameworks by providing developers with a set of tools and techniques that are as rich and as versatile as the frameworks themselves.

Appendix A

Extended Case Study

This appendix contains the final four APIs studied in the Spring case study described in Chapter 6. While not as interesting as the four show in Chapter 6, they are included for completeness. The quantitative results from the analysis are listed in Table 6.4.

A.1 Returning a ModelAndView with the errors map

Recall that Section 6.4.2 presented an example constraint about how to properly return a `Model-AndView` object from the `onSubmit` method. In the study, I found two other threads that were about a related constraint.

In thread 39209 [93], the user "senthilnathan74" was having problems getting the right model data returned. Ze wanted to return the `errors.getModel()` map as the model, as seen in Listing A.1, yet the view was throwing an exception when attempting to access the model map. The problem with hir code is that it is using the wrong constructor; this constructor will create a *new* map with a single key-value pair as given by the last two parameters. Instead, ze should have used the constructor that takes a `Map`, as shown in Listing A.2

In another thread [49], the user "gurnard" was instructed by "Colin Yates" to "add `errors.getModel()` to the `ModelAndView` you return from `onSubmit`." "gurnard"'s response was in Listing A.3, which also doesn't work, as it will add the `errors.getModel()` object as a value in the map rather than

Listing A.1: Incorrect way of creating a new `ModelAndView`.

```
1  protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response,
2                     Object command, BindException errors) throws Exception {
3    AccountForm accountForm = (AccountForm) command;
4    ...
5    ModelAndView mav = new ModelAndView(getSuccessView(), "account", errors.getModel());
6    return mav;
7  }
```

Listing A.2: Correct way to create a new `ModelAndView` with `errors.getModel()`.

```
1  protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response,
2                     Object command, BindException errors) throws Exception {
3    AccountForm accountForm = (AccountForm) command;
4    ...
5    ModelAndView mav = new ModelAndView(getSuccessView(), errors.getModel());
6    return mav;
7  }
```

Listing A.3: Another incorrect way of creating a new `ModelAndView`.

```
1  protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response,
2                     Object command, BindException errors) throws Exception {
3    AccountForm accountForm = (AccountForm) command;
4    ...
5    ModelAndView mav = new ModelAndView(getSuccessView(), "account", accountForm);
6    mav.addObject("errors", errors.getModel());
7    return mav;
8  }
```

adding all the items within it. Instead, ze should have used `addAllObjects()`, as seen in Listing A.5.

To specify these constraints, I first use an effect to mark `Maps` that are returned from a call to `errors.getModel()` as bound models, as seen in Listing A.4. Then, I create a constraints to prevent these methods from being called with a bound model. These constraints will allow Listing A.2 and A.5 to pass, but they will produce warnings from all three variants for Listings A.1 and A.3.

A.2   Using Web Flow Actions

One of the major sub-frameworks of Spring is the Web Flow framework. While many websites allow the user to navigate anywhere they like, certain series of actions in a web application have a specific path, or *flow*, that a user must follow. For example, the checkout process on many websites requires that users perform certain actions in a certain order. Spring Web Flow (SWF) allows programmers to define appropriate the appropriate paths that a user may take. These flows may branch depending on user input, and they may call to sub-flows.

Listing A.6 shows a simple flow where a user can attempt to login; if the login fails, it redirects back to the login page. Such a flow could be called by other flows to check if a user is logged in. For this flow to work, there must be beans that represent the action that is taken at each of these steps (ie: Lines 10 and 17). These beans must implement the `Action` interface or extend from a class which implements this interface, such as the `FormAction` class. Listing A.7 shows the beans that are used by this flow.

As described, this is a straightforward constraint. All beans referenced by an `action` tag in the flow must exist in the `ApplicationContext` and they must be a subtype of `Action`. However,

Listing A.4: Specifications

```
1  public class BindException extends Exception implements BindingResult
2      @BoundModel(target, result)
3       public Map getModel() {...}
4      ...
5  }
6
7  @Constraint(
8    op="ModelAndView(String view, String key, Object value)",
9    trg="BoundModel(errors, value)",
10   req="FALSE"
11 )
12 @Constraint(
13   op="ModelAndView.addObject(String key, Object object) : ModelAndView",
14   trg="BoundModel(errors, object)",
15   req="FALSE"
16 )
17 public class ModelAndView {...}
```

Listing A.5: Correct way of creating a new `ModelAndView` with a single key-value pair.

```
1  protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response,
2                    Object command, BindException errors) throws Exception {
3      AccountForm accountForm = (AccountForm) command;
4      ...
5      ModelAndView mav = new ModelAndView(getSuccessView(), "account", accountForm);
6      mav.addAllObjects(errors.getModel());
7      return mav;
8  }
```

Listing A.6: A simple example of a flow to log in to a system.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-1.0.xsd">

    <start-state idref="checkLogin" />

    <action-state id="checkLogin">
        <action bean="checkStudentLoggedInAction"/>
        <transition on="success" to="finish" />
        <transition on="error" to="enterLogin" />
    </action-state>

    <view-state id="enterLogin" view="details">
       <render-actions>
         <action bean="loginAction"/>
       </render-actions>
       <transition on="enter" to="validateStudentLogin" />
    </view-state>

    <action-state id="validateStudentLogin">
        <action bean="loginAction"/>
        <transition on="success" to="finish" />
        <transition on="error" to="enterLogin" />
    </action-state>

    <end-state id="finish"/>
</flow>
```

Listing A.7: Beans for the flow in Listing A.6

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="checkStudentLoggedInAction" class="org.springframework.webflow.action.Action"/>

   <bean id="loginAction" class="org.springframework.webflow.action.FormAction">
      <property name="formObjectClass" value="StudentLoginInfo"/>
      <property name="validator">
         <bean class="studentValidator"/>
      </property>
   </bean>

   <bean id="studentValidator" class="StudentLoginValidator"/>
</beans>
```

there is a very subtle mistake that a developer can make.

In thread 38940 [87], the developer "raydawg" was working with an application that uses both the Spring framework and the Struts framework. As described in Chapter 6, Spring is meant to work alongside many other frameworks, and is completely compatible with Struts, another common web application framework. This developer created a web flow similar to the one in Listing A.6 and referenced their own version of the `loginAction`:[1]

```
1  <bean id="loginAction" class="edu.ucr.c3.rsvp.controller.students.Login"/>
```

When the developer ran this flow, the framework produced the following error:

```
org.springframework.beans.factory.BeanNotOfRequiredTypeException:
  Bean named 'loginAction' must be of type
    [org.springframework.webflow.execution.Action],
  but was actually of type
    [edu.ucr.c3.rsvp.controller.students.Login]
```

This was very confusing for the developer; ze understood perfectly well that the `loginAction` must extend from `Action`. In fact, ze posted the code in Listing A.8 on the forum, to show that `Login` extended from the right classes.

The user "jeremyg484" discovered the problem:

> *It seems you are confusing a Struts action with an SWF action. FlowAction is SWF's integration point for Struts that is meant to launch or resume a flow. It is a Struts action and is to be configured in your struts-config. The action specified in your action-state on the other hand is an SWF action, and as you currently have it defined it must be an implementation of org.springframework.webflow.execution.Action as the error message states.*

In other words, while `FlowAction` is a class provided by Spring, it extends from the *Struts* `Action` interface, not the *Spring* `Action` interface!

To specify this constraint, we will use the Context relationship in Section 6.4.1 and a new relationship, Action(String) to represent the name of a bean which must be an action. The same XQuery from Section 6.4.1 will retrieve the Context relationship from the bean file (Listing 6.2), and the XQuery in Listing A.9 will retrieve the Action relationship from the flow file. In the case study, I found that certain relationships, like Context, were reused across many constraints.

The constraint itself is very simple, as shown in Listing A.10. It use the "XML" operator to check the declarative files before processing any Java files to ensure that they are consistent. When it finds an Action, it ensures that this action name was declared in the context with the right type.

A.3 Serializing Flow Objects

Spring Web Flow allow developers to create objects that are used throughout the flow. These objects are called "flow variables" and are defined in the flow file; Listing A.11 provides an example

---

[1]Yes, that package shows that this comes from a developer at UC Riverside. It's most interesting what you can learn from package names on public forums!

Listing A.8: Code posted by "raydawg" in [87].

```
1    public class Login extends RSVPAction {
2
3        public ActionForward executeRSVPApp(ActionMapping mapping, ActionForm form,
4              HttpServletRequest req, HttpServletResponse resp, HttpSession sess) throws Exception {
5
6          ActionForward forward = null;
7
8                     ....some database logic, etc....
9
10         return forward;
11       }//executeFRSApp
12   }
13
14   public abstract class RSVPAction extends FlowAction {
15
16       public RSVPAction() {
17         super();
18       }
19
20       /**
21        * Do a security check and only call the executeFRSApp method if
22        * it passes.
23        */
24       public final ActionForward execute(ActionMapping mapping, ActionForm form,
25             HttpServletRequest req, HttpServletResponse resp) throws Exception {
26
27         .....some code.....
28         return forward;
29       }//execute
30   }
```

Listing A.9: XQuery to retrieve the Action relationship

```
 1  declare namespace sf="http://www.springframework.org/schema/webflow";
 2  declare namespace fusion="http://code.google.com/p/fusion";
 3  declare variable $doc as xs:string external;
 4
 5  for $state in doc($doc)/sf:flow/sf:view-state
 6  for $action in $state/sf:render-actions/sf:action
 7  return <Relationship name="Action" effect="ADD">
 8          <Object name ="{data($action/@bean)}" type="java.lang.String"/>
 9       </Relationship>
10
11  for $state in doc($doc)/sf:flow/sf:view-state
12  for $action in $state/sf:transition/sf:action
13  return <Relationship name="Action" effect="ADD">
14          <Object name ="{data($action/@bean)}" type="java.lang.String"/>
15       </Relationship>
16
17  for $state in doc($doc)/sf:flow/sf:action-state
18  for $action in $state/sf:action
19  return <Relationship name="Action" effect="ADD">
20          <Object name ="{data($action/@bean)}" type="java.lang.String"/>
21       </Relationship>
```

Listing A.10: Constraint to check that all actions are actually an `Action`.

```
 1  @Constraint(
 2     op="XML",
 3     trg="Action(name)",
 4     req="Context(name, action, context) AND action instanceof Action"
 5  )
```

Listing A.11: A flow with a variable, example from [113]

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <flow xmlns="http://www.springframework.org/schema/webflow"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xsi:schemaLocation="http://www.springframework.org/schema/webflow
5                         http://www.springframework.org/schema/webflow/spring-webflow-1.0.xsd">
6
7      <var name="customer" class="com.springinaction.pizza.domain.Customer" scope="flow"/>
8
9      <start-state idref="askForPhoneNumber" />
10
11     <view-state id="askForPhoneNumber" view="phoneNumberForm">
12         <transition on="submit" to="lookupCustomer" />
13     </view-state>
14
15     <action-state id="lookupCustomer">
16         <action bean="lookupCistomerAction"/>
17         <transistion on="success" to="checkDeliveryArea"/>
18         <transition on-exception="com.springinaction.pizza.service.CustomerNotFoundException"
19                     to="addNewCustomer"/>
20     </action-state>
21
22     <decision-state id="checkDeliveryArea">
23         <if test="{$flowScope.customer.inDeliveryArea}"
24             then="finish"
25             else="warnNoDeliveryAvailable"/>
26     </decision-state>
27
28     <view-state id="addNewCustomer" ... />
29
30     <view-state id="warnNoDeliveryAvailable" ... />
31
32     <end-state id="finish" />
33   </flow>
```

of a flow variable being defined (line 7) and used (line 23). There are four possible "scopes" for a flow variable: request, flash, flow, and conversation. The scope defines the lifetime of the flow variable. For example, a request variable only lasts for the length of a single request from the user, while a flow variable will last for the entire flow but is not accessible in sub-flows. The framework controls the creation and destruction of these objects.

In some scopes, like flash and flow, the framework must be able to store the object across requests from the user. To do this, it serializes the object. This means that there is a hidden constraint: flow objects with a flash or flow scope must implement `Serializable`. If this is not the case, the framework will throw an exception at the point when it attempts to serialize the object.

To specify this, we first need to be aware of these flow variables that are declared in the flow configuration file. Listing A.12 retrieves two unary relationships that represent whether an object is a FlowVariable or a FlashVariable. The constraint specification itself runs after all the

Listing A.12: XQuery to retrieve the FlowVariable and FlashVariable relationships

```
1  declare namespace sf="http://www.springframework.org/schema/webflow";
2  declare namespace fusion="http://code.google.com/p/fusion";
3  declare variable $doc as xs:string external;
4
5  for $var in doc($doc)/sf:flow/sf:var
6  where $var/@scope = "flow"
7  return <Relationship name="FlowVariable" effect="ADD">
8         <Object name ="{data($var/@name)}" type="{data($var/@class)}"/>
9      </Relationship>
10
11 for $var in doc($doc)/sf:flow/sf:var
12 where $var/@scope = "flash"
13 return <Relationship name="FlashVariable" effect="ADD">
14         <Object name ="{data($var/@name)}" type="{data($var/@class)}"/>
15      </Relationship>
```

Listing A.13: Constraint to check that all flow and flash variables are `Serializable`.

```
1  @Constraint(
2      op="XML",
3      trg="FlowVariable(bean) OR FlashVariable(bean)",
4      req="bean instanceof Serializable"
5  )
```

Listing A.14: Using a `FormAction` in a single `view-state`

```
1   <flow xmlns="http://www.springframework.org/schema/webflow"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:schemaLocation="http://www.springframework.org/schema/webflow
4                      http://www.springframework.org/schema/webflow/spring-webflow-1.0.xsd">
5
6     <start-state idref="enterCriteria"/>
7
8     <view-state id="enterCriteria" view="searchCriteria">
9       <render-actions>
10          <action bean="formAction" method="setupForm"/>
11      </render-actions>
12      <transition on="search" to="displayResults">
13          <action bean="formAction" method="bindAndValidate"/>
14      </transition>
15    </view-state>
16      ...
17  </flow>
```

XML is loaded, and it verifies that all objects that are a FlowVariable or FlashVariable implement `Serializable`, as seen in Listing A.13.

It is interesting that this constraint takes exactly the same form as the constraint in Section A.2. This makes sense; both are checking that an object declared in XML has the right Java type. If XML was aware of these types, or if a custom typed configuration language was used instead, neither of these constraints would be necessary because they would be built into the typechecker. While Fusion can be used to encode a typesystem, it is certainly not the ideal way of doing so.

A.4   The FormAction lifecycle

In the same way that Spring provided a `Controller` hierarchy, it also provides an `Action` hierarchy with reusable subclasses for common tasks. The `FormAction` is an `Action` that represents a user's submitted data to a form, or set of forms across a flow, and works analogously to the `SimpleFormController`.

Using a `FormAction` is a little more complex though. While `SimpleFormController` ensures that all callbacks happen in the right order, `FormAction` depends on the programmer to make the callbacks for it within the XML flow. Listing A.14 provides an example of such a file. In this example, the programmer sets up the `FormAction` in the state "enterCriteria" (line 10) and then binds and validates it at the same time in the transition out of the state (line 13). Listing A.15 shows how these can be split up across multiple states; this example sets up the `FormAction` on entry to the "enterCustomerDetails" state, binds it on the "submit" transition, and validates it in the "processDetails" state.

Notice that Web Flow provides a great deal of flexibility; we can perform other actions between these states, skip the user ahead based upon entered data, or even cancel the entire flow at any time. This flexibility comes at the cost of usability of the API though. The programmer must

Listing A.15: Using a `FormAction` in multiple states, based on code from [96]

```
1  <flow xmlns="http://www.springframework.org/schema/webflow"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://www.springframework.org/schema/webflow
4                    http://www.springframework.org/schema/webflow/spring-webflow-1.0.xsd">
5
6   <start-state idref="enterCustomerDetails"/>
7
8   <view-state id="enterCustomerDetails" view="cutsomerRegisterForm">
9     <entry-actions>
10        <action bean="customerRegisterAction" method="setupForm"/>
11     </entry-actions>
12     <transition on="submit" to="processDetails">
13        <action bean="customerRegisterAction" method="bind"/>
14     </transition>
15   </view-state>
16
17   <action-state id="processDetails">
18     <action bean="customerRegisterAction" method="validate"/>
19      <transition on="success" to="enterEnquiryDetails"/>
20     <transition on="error" to="enterCustomerDetails"/>
21   </action-state>
22     ...
23  </flow>
```

still respect the unwritten rules about the order in which things may be called. In the case study, three programmers [34, 81, 96] did not set up the `FormAction` before binding it. This caused unusual problems, including not transitioning in exception conditions (results in not catching the exception), not having the model data available in the view (results in an exception from the view), and missing property editors that cause the view to display strangely.

To describe the constraint that a `FormAction` must be set up at *some* point before being bound, we will need the following four relationships:

- SetupAction(String, FormAction) provides the name of the state that sets up a `FormAction`.

- BindAction(String, FormAction) provides the name of the state that binds a `FormAction`.

- Transition(String, String, String) describes the transition step from one state to another state.

- Path(String, String) represents the existence of a path from one state to another through Transitions.

The XQuery to retrieve the first three relationships is shown in Listing A.16. The Path relationship is more unusual. This relationship represents the transitive closure on the Transition relationship and is created through use of the @Infer specs shown in Listing A.17.

Again, the constraint itself is simple: we specify that the XML must ensure that if a binding call is made on a `FormAction`, then a setup call must have occurred sometime in advance. This constraint is shown in Listing A.18.

Listing A.16: XQuery to retrieve the SetupAction, BindAction and Transition relationships

```
1   declare namespace sf="http://www.springframework.org/schema/webflow";
2   declare namespace fusion="http://code.google.com/p/fusion";
3   declare variable $doc as xs:string external;
4
5   for $state in doc($doc)/sf:flow/sf:view-state
6   for $action in $state/sf:render-actions/sf:action
7   where $action/@method = "setupForm"
8   return
9   <Relationship name="SetupAction" effect="ADD">
10     <Object name ="{data($state/@id)}" type="java.lang.String"/>
11     <Object name ="{data($action/@bean)}" type="org.springframework.webflow.action.FormAction"/>
12  </Relationship>
13
14  for $state in doc($doc)/sf:flow/sf:view-state
15  for $action in $state/sf:transition/sf:action
16  where $action/@method = "bindAndValidate"
17  return
18  <Relationship name="BindAction" effect="ADD">
19     <Object name ="{data($state/@id)}" type="java.lang.String"/>
20     <Object name ="{data($action/@bean)}" type="org.springframework.webflow.action.FormAction"/>
21  </Relationship>
22
23  for $state in doc($doc)/sf:flow/sf:view-state
24  for $action in $state/sf:transition/sf:action
25  where $action/@method = "bind"
26  return
27  <Relationship name="BindAction" effect="ADD">
28     <Object name ="{data($state/@id)}" type="java.lang.String"/>
29     <Object name ="{data($action/@bean)}" type="org.springframework.webflow.action.FormAction"/>
30  </Relationship>
31
32  for $state in doc($doc)/sf:flow/sf:view-state
33  for $trans in $state/sf:transition
34  return
35  <Relationship name="Transition" effect="ADD">
36     <Object name ="{data($state/@id)}" type="java.lang.String"/>
37     <Object name ="{data($trans/@on)}" type="java.lang.String"/>
38     <Object name ="{data($trans/@to)}" type="java.lang.String"/>
39  </Relationship>
```

Listing A.17: Specifications to infer a path between states.

```
1  @Infer(
2      trg="Transition(pState, t, state) AND Transition(state, s, nState)",
3      eff={"Path(pState, nState)"}
4  )
5  @Infer(
6      trg="Transition(pState, t, nState)",
7      eff={"Path(pState, nState)"}
8  )
```

Listing A.18: Specifications to enforce that setup always occurs sometime before binding.

```
1  @Constraint(
2      op="XML",
3      trg="BindAction(state, form)",
4      req="SetupAction(pState, form) AND Path(pState, state)"
5  )
```

This constraint shows one of the interesting differences between the three variants of the analysis. Recall from Chapter 5 that while the trigger predicate will bind all variables with a universal quantifier, the requires predicate uses either a universal or existential depending on the variant. This issue only becomes relevant in cases like Listing A.18, where a variable is used only in the requires predicate (`pState`). Therefore, for the complete variant, this constraint reads "if a state binds a form, then *some* prior state must have setup the form." On the other hand, the sound variant checks that "if a state binds a form, then *all* prior states must have setup the form." Given this, it is unsurprising that the sound variant always gives a warning in practice.

Appendix B

Formalism

This appendix formally presents the abstract grammar and semantics of the specifications and analysis. The first section provides the grammar, the following sections define several operators and functions on elements of the grammar, and the final section presents the inference rules that define the formal semantics. In this appendix, I will be using the following typographical notations:

- an overbar ($\bar{x}$, $\bar{\ell}$, $\bar{y} : \bar{\tau}$) represents an ordered list. $|\bar{x}|$ gives the length of the list.

- braces ($\{\ell\}$, $\{cons\}$, $\{P \Downarrow \overline{Q}\}$) represents an unordered set.

- braces with an arrow ($\{y \mapsto x\}$) represents an unordered map with unique keys which can be used to retrieve values from the map. dom and rng functions can be used to access the domain or range of a map.

- braces with two semicolons ($\{A; B; C\}$) represent a set of triples. Projection can be used ($\{A; B; C\}.B$) to access a set with a single element of the triple.

- $\varnothing$ represents an empty list, set, or map.

- sets and maps can be created with set construction ($\sigma = \{y \mapsto \ell \mid X(y, x) = \ell\}$)

B.1 Abstract Grammar

Listing B.1 describes the abstract grammar of Fusion. In this grammar, I use the following special variables:

- x represents a source variable

- y represents a specification variable, where the values target and result have special meanings

- m represents a method name

131

- rel represents a relation name

- τ represents a type

- ℓ represents a label for an abstraction of a runtime object

A constraint is represented with cons, which has the five parts described in Chapters 4 and 5. P is a logical predicate on relationship predicates R, which are across specification variables y. For this formalism, the only atomic predicates are relationship predicates, but this is easily extended. M, N, T and R are analogous to P, Q, A, and S, but they are across object labels ℓ instead of specification variables. R is an actual relationship across abstractions of objects as described in Chapter 4.

Source instructions are represented in three address code with instr, and the specifications to describe them are shown as op. Only four instructions and operations are shown, but this is also easily extended in the obvious manner.

The flow lattice is a map of relationships to ternary values. There is also a "delta lattice" that represents the effects that should be made to the flow lattice. This δ uses a seven-point lattice with elements E, where bot represents "the constraint does not apply" and $*$ represents "the constraint applies, but no change was specified." I will refer to bot as the "no effect" and $*$ as the "ignore effect".

The next pieces of the grammar represent the bindings from specification variables to source variables and object labels. It also includes the grammar needed for making strong updates to variables.

The last pieces of the grammar are all environments that will be used. As before, $\mathcal{B}$ and $\mathcal{A}$ are the boolean constant propagation lattice and the points-to lattice respectively. $\mathcal{R}$, $\mathcal{C}$, and $\mathcal{I}$ are the sets of Fusion relations, constraint specifications, and inference specifications to use in the analysis.

constraint	cons	::=	$op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}_{eff}; P_{rst}$
predicate	P	::=	$P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid Q \mid \text{true} \mid \text{false}$
negation predicate	Q	::=	$\neg A \mid A$
atomic predicate	A	::=	$S \mid S/y \mid \ldots$
relation predicate	S	::=	$\text{rel}(\bar{y})$
relationship logic	M	::=	$M_1 \wedge M_2 \mid M_1 \vee M_2 \mid M_1 \implies M_2 \mid N \mid \text{true} \mid \text{false}$
negation relationship	N	::=	$\neg T \mid T$
atomic relationship	T	::=	$R \mid R/\ell \mid \ldots$
relationship	R	::=	$\text{rel}(\bar{\ell})$
source instruction	instr	::=	$x_{ret} = x_{this}.m(\bar{x}) \mid x_{ret} = \text{new } \tau(\bar{x}) \mid$
			$\text{return } x_{ret}(x_{this}.m(\bar{x})) \mid \text{begin}(x.m(\bar{x})) \mid \ldots$
instruction signature	op	::=	$\tau_{this}.m(\bar{\tau}\,\bar{y}) : \tau_{ret} \mid \text{new } \tau(\bar{\tau}\,\bar{y}) \mid$
			$\text{eom}(\tau_{this}.m(\bar{\tau}\,\bar{y}) : \tau_{ret}) \mid \text{bom}(\tau_{this}.m(\bar{\tau}\,\bar{y})) \mid \ldots$
flow lattice	ρ	::=	$\{R \mapsto t\}$
ternary logic	t	::=	$\text{True} \mid \text{False} \mid \text{Unknown}$
delta lattice	δ	::=	$\{R \mapsto E\}$
delta lattice elements	E	::=	$\text{unknown} \mid \text{true} \mid \text{false} \mid \text{true} * \mid \text{false} * \mid * \mid \text{bot}$
variable binding	β	::=	$\{y \mapsto x\}$
substitution	σ	::=	$\{y \mapsto \ell\}$
set of substitutions	Σ	::=	$\{\sigma\}$
spec updates	α	::=	$\{y \mapsto \{\ell\}\}$
source updates	γ	::=	$\{x \mapsto \{\ell\}\}$
bool constants lattice	$\mathcal{B}$	::=	$\{\ell \mapsto t\}$
alias lattice	$\mathcal{A}$	::=	$< \Gamma_\ell; \mathcal{L} >$
aliases	$\mathcal{L}$	::=	$\{x \mapsto \{\ell\}\}$
location types	Γ_ℓ	::=	$\{\ell : \tau\}$
spec variable types	Γ_y	::=	$\{y : \tau\}$
relation type	$\mathcal{R}$	::=	$\{\text{rel} \mapsto \bar{\tau}\}$
constraints	$\mathcal{C}$	::=	$\{\text{cons}\}$
relation inference rules	$\mathcal{I}$	::=	$\{P \Downarrow \bar{Q}\}$

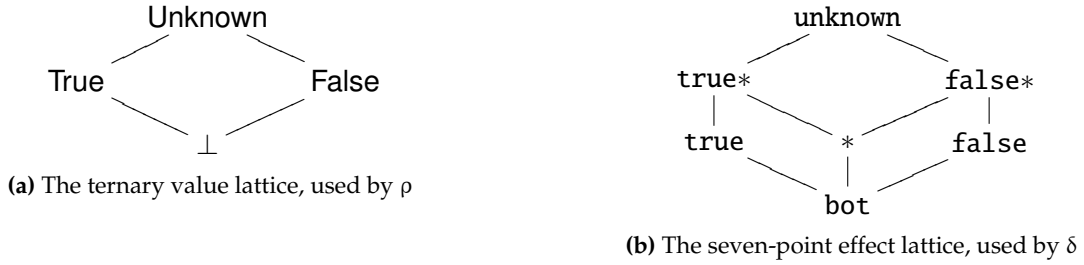**Figure B.1:** Abstract grammar of Fusion

B.2   Operations on lattices

There are two lattices used in the semantics. The flow lattice ρ is the lattice used by the flow analysis. ρ is a tuple lattice of relationships to ternary values, which are in the lattice shown by Figure B.2a. The effect lattice δ is only used internally in the Fusion semantics. It is also a tuple lattice, but it maps relationships to the seven-point effect lattice in Figure B.2b.

Both of these sub-lattices have the expected lattice operations ($\sqsubseteq$ and $\sqcup$), plus there are four additional operators as seen in Figures B.3 and B.4 .

- The equality join operator $\boxminus$ is similar to the $\sqcup$ operator, but it recombines `true` with `true`∗ and `false` with `false`∗

- The override operator $\boxminus$ allows one effect to override the other, unless it is `bot` or ∗.

- The polarize operator $\overset{\uparrow}{*}$ moves the `true` and `false` elements to `true`∗ and `false`∗ respectively. This is almost the same as $E \sqcup *$, except that `bot` remains where it is.

- The change operator $\Leftarrow$ makes the effect prescribed in E onto the ternary value t.

These operators on the sub-lattices are used in the expected way on the parent lattices. Figure B.5 shows the operators for δ, and Figure B.6 shows the operators for ρ.

(a) The ternary value lattice, used by ρ

(b) The seven-point effect lattice, used by δ

Figure B.2: The sub lattices used by ρ and δ

$$\boxed{E_l \sqcup\!\!\!= E_r = E'}$$

$$
\begin{aligned}
E \sqcup\!\!\!= bot &= E \\
bot \sqcup\!\!\!= E &= E \\
E \sqcup\!\!\!= * &= E \\
* \sqcup\!\!\!= E &= E \\
E \sqcup\!\!\!= E &= E
\end{aligned}
$$

$$
\begin{aligned}
true \sqcup\!\!\!= true* &= true \\
true * \sqcup\!\!\!= true &= true \\
false \sqcup\!\!\!= false* &= false \\
false * \sqcup\!\!\!= false &= false
\end{aligned}
$$

$$
\begin{aligned}
true \sqcup\!\!\!= false &= unknown \\
false \sqcup\!\!\!= true &= unknown \\
true * \sqcup\!\!\!= false &= unknown \\
false \sqcup\!\!\!= true* &= unknown \\
false * \sqcup\!\!\!= true &= unknown \\
true \sqcup\!\!\!= false* &= unknown \\
true * \sqcup\!\!\!= false* &= unknown \\
false * \sqcup\!\!\!= true* &= unknown
\end{aligned}
$$

$$
\begin{aligned}
E \sqcup\!\!\!= unknown &= unknown \\
unknown \sqcup\!\!\!= E &= unknown
\end{aligned}
$$

Figure B.3: Equality join operator on E

$\boxed{E \Leftarrow E' = E''}$

$$
\begin{aligned}
E \Leftarrow \texttt{bot} &= E \\
E \Leftarrow \ast &= E \\
E \Leftarrow \texttt{true} &= \texttt{true} \\
E \Leftarrow \texttt{true}\ast &= \texttt{true}\ast \\
E \Leftarrow \texttt{false} &= \texttt{false} \\
E \Leftarrow \texttt{false}\ast &= \texttt{false}\ast \\
E \Leftarrow \texttt{unknown} &= \texttt{unknown}
\end{aligned}
$$

$\boxed{\overset{\uparrow}{\ast} E = E'}$

$$
\begin{aligned}
\overset{\uparrow}{\ast} \texttt{false} &= \texttt{false}\ast \\
\overset{\uparrow}{\ast} \texttt{true} &= \texttt{true}\ast \\
\overset{\uparrow}{\ast} \ast &= \ast \\
\overset{\uparrow}{\ast} \texttt{bot} &= \texttt{bot} \\
\overset{\uparrow}{\ast} \texttt{unknown} &= \texttt{unknown} \\
\overset{\uparrow}{\ast} \texttt{true}\ast &= \texttt{true}\ast \\
\overset{\uparrow}{\ast} \texttt{false}\ast &= \texttt{false}\ast
\end{aligned}
$$

$\boxed{t \Leftarrow E = t'}$

$$
\begin{aligned}
t \Leftarrow \texttt{bot} &= t \\
t \Leftarrow \ast &= t \\
\text{False} \Leftarrow \texttt{false}\ast &= \text{False} \\
\text{True} \Leftarrow \texttt{false}\ast &= \text{Unknown} \\
\text{Unknown} \Leftarrow \texttt{false}\ast &= \text{Unknown} \\
\text{True} \Leftarrow \texttt{true}\ast &= \text{True} \\
\text{False} \Leftarrow \texttt{true}\ast &= \text{Unknown} \\
\text{Unknown} \Leftarrow \texttt{true}\ast &= \text{Unknown} \\
t \Leftarrow \texttt{false} &= \text{False} \\
t \Leftarrow \texttt{true} &= \text{True} \\
t \Leftarrow \texttt{unknown} &= \text{Unknown}
\end{aligned}
$$

Figure B.4: Operations on the elements of the relationship lattice, E

$\boxed{\delta \sqsubseteq \delta'}$

$$\frac{}{\varnothing \sqsubseteq \delta}{}^{(\sqsubseteq-\varnothing)} \qquad\qquad \frac{\delta^c \sqsubseteq \delta^a \qquad E^c \sqsubseteq E^a}{R \mapsto E^c, \delta^c \sqsubseteq R \mapsto E^a, \delta^a}{}^{(\sqsubseteq-\delta)}$$

$\boxed{\delta \sqcup \delta' = \delta''}$

$$\frac{}{\varnothing \sqcup \varnothing = \varnothing}{}^{(\sqcup-\varnothing)} \qquad\qquad \frac{\delta_l \sqcup \delta_r = \delta' \qquad E_l \sqcup E_r = E'}{R \mapsto E_l, \delta_l \sqcup R \mapsto E_r, \delta_r = R \mapsto E', \delta'}{}^{(\sqcup-\delta)}$$

$\boxed{\delta \mathrel{\boxminus} \delta' = \delta''}$

$$\frac{}{\varnothing \mathrel{\boxminus} \varnothing = \varnothing}{}^{(\text{EQJOIN}-\varnothing)} \qquad\qquad \frac{\delta_l \mathrel{\boxminus} \delta_r = \delta' \qquad E_l \mathrel{\boxminus} E_r = E'}{R \mapsto E_l, \delta_l \mathrel{\boxminus} R \mapsto E_r, \delta_r = R \mapsto E', \delta'}{}^{(\text{EQJOIN}-\delta)}$$

$\boxed{\delta \mathrel{\boxleftarrow} \delta' = \delta''}$

$$\frac{}{\varnothing \mathrel{\boxleftarrow} \varnothing = \varnothing}{}^{(\text{OVERRIDE}-\varnothing)} \qquad\qquad \frac{\delta_l \mathrel{\boxleftarrow} \delta_r = \delta' \qquad E_l \mathrel{\boxleftarrow} E_r = E'}{R \mapsto E_l, \delta_l \mathrel{\boxleftarrow} R \mapsto E_r, \delta_r = R \mapsto E', \delta'}{}^{(\text{OVERRIDE}-\delta)}$$

$\boxed{\overset{\uparrow}{*}\, \delta = \delta'}$

$$\frac{}{\overset{\uparrow}{*}\, \varnothing = \varnothing}{}^{(\text{POLAR}-\varnothing)} \qquad\qquad \frac{\overset{\uparrow}{*}\, \delta = \delta' \qquad \overset{\uparrow}{*}\, E = E'}{\overset{\uparrow}{*}\, R \mapsto E, \delta = R \mapsto E', \delta'}{}^{(\text{POLAR}-\delta)}$$

Figure B.5: Operations on the change lattice, δ

$\boxed{\rho \sqsubseteq \rho'}$

$$\frac{}{\varnothing \sqsubseteq \rho}{}^{(\sqcup - \varnothing)} \qquad\qquad \frac{\rho^c \sqsubseteq \rho^a \qquad t^c \sqsubseteq t^a}{R \mapsto t^c, \rho^c \sqsubseteq R \mapsto t^a, \rho^a}{}^{(\sqcup - \rho)}$$

$\boxed{\rho \sqcup \rho' = \rho''}$

$$\frac{}{\varnothing \sqcup \varnothing = \varnothing}{}^{(\sqcup - \varnothing)} \qquad\qquad \frac{\rho_l \sqcup \rho_r = \rho' \qquad t_l \sqcup t_r = t'}{R \mapsto t_l, \rho_l \sqcup R \mapsto t_r, \rho_r = R \mapsto t', \rho'}{}^{(\sqcup - \rho)}$$

$\boxed{\rho \Leftarrow \delta = \rho'}$

$$\frac{}{\varnothing \Leftarrow \varnothing = \varnothing}{}^{(\Leftarrow - \varnothing)} \qquad\qquad \frac{\rho \Leftarrow \delta = \rho' \qquad t \Leftarrow E = t'}{R \mapsto t, \rho \Leftarrow R \mapsto E, \delta = R \mapsto t', \rho'}{}^{(\Leftarrow - \rho)}$$

Figure B.6: Operations on the relationship lattice, ρ

B.3 Operations on specifications

Substitution of predicates is straightforward. Figure B.7 shows how given a P over specification variables and a substitution σ, we can create a predicate in the target language over object labels ℓ.

The semantics will need to be able to access the free variable that occur within each part of the constraint and the type that is expected. This will be represented by the specification typing environment $Γ_y$. Figure B.8 shows how the free variables are retrieved from the specifications. The constraint itself finds its free variables by combining the free variables of all the subparts. This operator must respect the types required by each part, as seen in Figure B.9. Notice that the semantics are that if two typing contexts have different types for a given y, then one must be a subtype of the other. Theoretically, this could be extended to allow for intersection types, and in fact the implementation of Fusion will allow this.

$$\boxed{P[\sigma] = M}$$

$$
\begin{aligned}
(P_1 \wedge P_2)[\sigma] &= P_1[\sigma] \wedge P_2[\sigma] \\
(P_1 \vee P_2)[\sigma] &= P_1[\sigma] \vee P_2[\sigma] \\
(P_1 \implies P_2)[\sigma] &= P_1[\sigma] \implies P_2[\sigma] \\
\mathsf{true}[\sigma] &= \mathsf{true} \\
\mathsf{false}[\sigma] &= \mathsf{false} \\
(\neg A)[\sigma] &= \neg A[\sigma] \\
(\mathsf{rel}(\bar{y})/y_{\mathsf{test}})[\sigma] &= \mathsf{rel}(\bar{y})[\sigma]/\sigma(y_{\mathsf{test}}) \\
\mathsf{rel}(\bar{y})[\sigma] &= \mathsf{rel}(\bar{y}[\sigma]) \\
(y, \bar{y})[\sigma] &= \sigma(y), \bar{y}[\sigma]
\end{aligned}
$$

Figure B.7: Substitutions on specifications.

$\boxed{\mathsf{FV}(\mathsf{cons}) = \Gamma_\mathsf{y}}$

$$\mathsf{FV}(\mathsf{op} : \mathsf{P}_{\mathsf{ctx}} \Rightarrow \mathsf{P}_{\mathsf{req}} \Downarrow \bar{\mathsf{Q}}_{\mathsf{eff}}; \mathsf{P}_{\mathsf{rst}}) \;=\; \mathsf{FV}(\mathsf{op}) \cup \mathsf{FV}(\mathsf{P}_{\mathsf{ctx}}) \cup \mathsf{FV}(\mathsf{P}_{\mathsf{req}}) \cup \mathsf{FV}(\overline{\mathsf{Q}}) \cup \mathsf{FV}(\mathsf{P}_{\mathsf{rst}})$$

$\boxed{\mathsf{FV}(\mathsf{P}) = \Gamma_\mathsf{y}}$

$$
\begin{aligned}
\mathsf{FV}(\mathsf{P}_1 \wedge \mathsf{P}_2) &= \mathsf{FV}(\mathsf{P}_1) \cup \mathsf{FV}(\mathsf{P}_2) \\
\mathsf{FV}(\mathsf{P}_1 \vee \mathsf{P}_2) &= \mathsf{FV}(\mathsf{P}_1) \cup \mathsf{FV}(\mathsf{P}_2) \\
\mathsf{FV}(\mathsf{P}_1 \implies \mathsf{P}_2) &= \mathsf{FV}(\mathsf{P}_1) \cup \mathsf{FV}(\mathsf{P}_2) \\
\mathsf{FV}(\mathsf{true}) &= \varnothing \\
\mathsf{FV}(\mathsf{false}) &= \varnothing \\
\mathsf{FV}(\neg A) &= \mathsf{FV}(A) \\
\mathsf{FV}(\mathsf{rel}(\bar{\mathsf{y}})/\mathsf{y}_{\mathsf{test}}) &= \mathsf{FV}(A), \mathsf{y}_{\mathsf{test}} : \mathsf{boolean} \\
\mathsf{FV}(\mathsf{rel}(\bar{\mathsf{y}})) &= \bar{\mathsf{y}} : \mathcal{R}(\mathsf{rel})
\end{aligned}
$$

$\boxed{\mathsf{FV}(\mathsf{instr}) = \Gamma_\mathsf{y}}$

$$
\begin{aligned}
\mathsf{FV}(\tau_{\mathsf{this}}.\mathsf{m}(\overline{\tau}\overline{\mathsf{y}}) : \tau_{\mathsf{ret}}) &= \mathsf{target} : \tau_{\mathsf{this}}, \mathsf{result} : \tau_{\mathsf{ret}}, \overline{\mathsf{y}} : \overline{\tau} \\
\mathsf{FV}(\,\mathsf{new}\ \tau(\overline{\tau}\overline{\mathsf{y}})) &= \mathsf{target} : \tau, \overline{\mathsf{y}} : \overline{\tau} \\
\mathsf{FV}(\mathsf{eom}\tau_{\mathsf{this}}.\mathsf{m}(\overline{\tau}\overline{\mathsf{y}}) : \tau_{\mathsf{ret}}) &= \mathsf{result} : \tau_{\mathsf{ret}}, \mathsf{target} : \tau_{\mathsf{this}} \\
\mathsf{FV}(\mathsf{bom}\tau_{\mathsf{this}}.\mathsf{m}(\overline{\tau}\overline{\mathsf{y}})) &= \mathsf{target} : \tau_{\mathsf{this}}, \overline{\mathsf{y}} : \overline{\tau}
\end{aligned}
$$

Figure B.8: Generating free variables from specifications

$\boxed{\Gamma_\mathsf{y} \cup \Gamma'_\mathsf{y} = \Gamma''_\mathsf{y}}$

$$\frac{}{\Gamma_\mathsf{y} \cup \varnothing = \Gamma_\mathsf{y}}{}^{(\cup - \varnothing)} \qquad\qquad \frac{\mathsf{y} \notin \mathsf{dom}(\Gamma^\mathsf{l}_\mathsf{y}) \quad \Gamma^\mathsf{l}_\mathsf{y} \cup \Gamma^\mathsf{r}_\mathsf{y} = \Gamma_\mathsf{y}}{\Gamma^\mathsf{l}_\mathsf{y} \cup \mathsf{y} : \tau, \Gamma^\mathsf{r}_\mathsf{y} = \mathsf{y} : \tau, \Gamma_\mathsf{y}}{}^{(\cup - \mathrm{NOTIN})}$$

$$\frac{\tau^\mathsf{l} <: \tau^\mathsf{r} \quad \Gamma^\mathsf{l}_\mathsf{y} \cup \Gamma^\mathsf{r}_\mathsf{y} = \Gamma_\mathsf{y}}{\mathsf{y} : \tau^\mathsf{l}, \Gamma^\mathsf{l}_\mathsf{y} \cup \mathsf{y} : \tau^\mathsf{r}, \Gamma^\mathsf{r}_\mathsf{y} = \mathsf{y} : \tau^\mathsf{l}, \Gamma_\mathsf{y}}{}^{(\cup - \mathrm{LEFTSUB})} \qquad \frac{\tau^\mathsf{r} <: \tau^\mathsf{l} \quad \Gamma^\mathsf{l}_\mathsf{y} \cup \Gamma^\mathsf{r}_\mathsf{y} = \Gamma_\mathsf{y}}{\mathsf{y} : \tau^\mathsf{l}, \Gamma^\mathsf{l}_\mathsf{y} \cup \mathsf{y} : \tau^\mathsf{r}, \Gamma^\mathsf{r}_\mathsf{y} = \mathsf{y} : \tau^\mathsf{r}, \Gamma_\mathsf{y}}{}^{(\cup - \mathrm{RIGHT-SUB})}$$

$\boxed{\Gamma_\mathsf{y} \subseteq \Gamma'_\mathsf{y}}$

$$\frac{\mathsf{dom}(\Gamma_\mathsf{y}) \subseteq \mathsf{dom}(\Gamma'_\mathsf{y}) \quad \forall \mathsf{y} : \tau \in \Gamma_\mathsf{y} \,.\, \Gamma'_\mathsf{y} <: \tau}{\Gamma_\mathsf{y} \subseteq \Gamma'_\mathsf{y}}{}^{(\subseteq - \Gamma_\mathsf{Y})}$$

Figure B.9: Operations on free variables

B.4 Points-to Operations

While I have defined the points-to lattice as $< \Gamma_\ell, \mathcal{L} >$, it actually has a third part, $\Gamma_\mathbf{x}$, that gives the types of the variables. However, as this is static information, this is always the same, regardless of whether we have an abstract or concrete heap. As it is only used in the operator matching rules, it will be elided.

Recall from Chapter 4 that $\mathcal{A}$ must always respect the abstraction from Theorem 6. Given this, the $\sqsubseteq$ operation on $\mathcal{A}$ must be given as shown in Figure B.10. Figure B.10 also shows the operation to make a strong update to $\mathcal{A}$ with the updates in γ.

When a constraint generates a strong update, it will initially be on α, a mapping on specification variables. This will eventually be converted into γ, a mapping on source variables, using the bindings from β. The $\sqsubseteq$ operator for α and γ is shown in Figure B.11, and the substitution using β is in Figure B.12.

$$\boxed{\mathcal{A} \sqsubseteq_{\mathcal{A}} \mathcal{A}'}$$

$$\frac{\mathrm{dom}(\mathcal{L}') = \mathrm{dom}(\mathcal{L})}{\mathrm{dom}(\Gamma'_\ell) \subseteq \mathrm{dom}(\Gamma_\ell) \quad \forall\, \ell' : \tau' \in \Gamma'_\ell.\ \tau' <: \Gamma_\ell(\ell') \quad \forall\, \mathbf{x}' \mapsto \bar{\ell}' \in \mathcal{L}'.\ \bar{\ell}' \subseteq \mathcal{L}(\mathbf{x}') \,\wedge\, \bar{\ell}' \neq \varnothing}{< \Gamma'_\ell; \mathcal{L}' >\sqsubseteq_{\mathcal{A}}< \Gamma_\ell; \mathcal{L} >}()(\sqsubseteq_{\mathcal{A}})$$

$$\boxed{\mathcal{A} \Leftarrow \gamma = \mathcal{A}'}$$

$$\frac{}{\mathcal{A} \Leftarrow \varnothing = \mathcal{A}}(\Leftarrow-\varnothing) \qquad \frac{< \Gamma_\ell; \mathcal{L} >\Leftarrow \gamma =< \Gamma'_\ell; \mathcal{L}' > \quad \mathbf{x} \in \mathrm{dom}(\mathcal{L}) \quad \{\ell\} \subseteq \mathcal{L}(\mathbf{x})}{< \Gamma_\ell; \mathcal{L} >\Leftarrow \mathbf{x} \mapsto \{\ell\}, \gamma =< \Gamma'_\ell; \mathcal{L}'[\mathbf{x} \mapsto \{\ell\}] >}(\Leftarrow-\mathrm{SET})$$

Figure B.10: Operations on the points-to lattice $\mathcal{A}$

$\boxed{\gamma \sqsubseteq \gamma'}$

$$\dfrac{}{\varnothing \sqsubseteq \gamma}(\sqsubseteq-\varnothing) \qquad\qquad \dfrac{\gamma^c \sqsubseteq \gamma^a \qquad \{\ell\}^c \subseteq \{\ell\}^a}{\mathbf{x} \mapsto \{\ell\}^c, \gamma^c \sqsubseteq \mathbf{x} \mapsto \{\ell\}^a, \gamma^a}(\sqsubseteq-\cancel{\varnothing})$$

$\boxed{\gamma_l \sqcup \gamma_r = \gamma'}$

$$\dfrac{}{\varnothing \sqcup \varnothing = \varnothing}(\sqcup-\varnothing) \qquad\qquad \dfrac{\gamma_l \sqcup \gamma_r = \gamma' \qquad \{\ell\}^l \cup \{\ell\}^r = \{\ell\}}{\mathbf{x} \mapsto \{\ell\}^l, \gamma_l \sqcup \mathbf{x} \mapsto \{\ell\}^r, \gamma_r = \mathbf{x} \mapsto \{\ell\}, \gamma'}(\sqcup-\cancel{\varnothing})$$

$\boxed{\alpha \sqsubseteq \alpha'}$

$$\dfrac{}{\varnothing \sqsubseteq \varnothing}(\sqsubseteq-\varnothing) \qquad\qquad \dfrac{\alpha^c \sqsubseteq \alpha^a \qquad \{\ell\}^c \subseteq \{\ell\}^a}{\mathbf{y} \mapsto \{\ell\}^c, \alpha^c \sqsubseteq \mathbf{y} \mapsto \{\ell\}^a, \alpha^a}(\sqsubseteq-\cancel{\varnothing})$$

Figure B.11: Precision of γ and α

$\boxed{\alpha[\beta] = \gamma}$

$$
\begin{aligned}
\varnothing[\beta] &= \varnothing \\
\mathbf{y} \mapsto \{\ell\}, \alpha[\beta] &= \mathbf{y} \mapsto \{\ell\}[\beta], \alpha[\beta]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{y} \mapsto \{\ell\}[\mathbf{y} \mapsto \mathbf{x}, \beta] &= \mathbf{x} \mapsto \{\ell\} \\
\mathbf{y}_1 \mapsto \{\ell\}[\mathbf{y}_2 \mapsto \mathbf{x}, \beta] &= \mathbf{y}_1 \mapsto \{\ell\}[\beta] \\
\mathbf{y} \mapsto \{\ell\}[\varnothing] &= \varnothing
\end{aligned}
$$

Figure B.12: Substitution on α

B.5   The Boolean Constant Propagation lattice

The Fusion analysis also relies on a boolean constant propagation analysis. Fusion assumes an abstraction of this lattice that maps object labels to ternary values and the expected precision operator $\sqsubseteq$ as shown in Figure B.13. Fusion uses this lattice when creating an effect based upon a relationship effect. Figure B.14 shows the rules for the function $value$, which will create a mapping $R \mapsto E$ based upon the lattice and an effect N.

$$\boxed{\mathcal{B} \sqsubseteq_{\mathcal{B}} \mathcal{B}'}$$

$$\frac{\text{dom}(\mathcal{B}^c) = \text{dom}(\mathcal{B}^a) \qquad \forall\, \ell : t \in \mathcal{B}^c.\; t \sqsubseteq \mathcal{B}^a(\ell)}{\mathcal{B}^c \sqsubseteq_{\mathcal{B}} \mathcal{B}^a}\;()(\sqsubseteq_{\mathcal{B}})$$

Figure B.13: Precision for the boolean constant propagation lattice

$$\boxed{value(\mathcal{B}; N) = R \mapsto E}$$

$$\frac{}{value(\mathcal{B}; R) = R \mapsto \texttt{true}}\,(\text{VAL}-\text{R}) \qquad\qquad \frac{}{value(\mathcal{B}; \neg R) = R \mapsto \texttt{false}}\,(\text{VAL}-\neg\text{R})$$

$$\frac{\mathcal{B}(\ell) = \textsf{True}}{value(\mathcal{B}; R/\ell) = R \mapsto \texttt{true}}\,(\text{VAL}-\text{T}-\text{TRUE}) \qquad \frac{\mathcal{B}(\ell) = \textsf{True}}{value(\mathcal{B}; \neg R/\ell) = R \mapsto \texttt{false}}\,(\text{VAL}-\neg\text{T}-\text{FALSE})$$

$$\frac{\mathcal{B}(\ell) = \textsf{False}}{value(\mathcal{B}; R/\ell) = R \mapsto \texttt{false}}\,(\text{VAL}-\text{T}-\text{FALSE}) \qquad \frac{\mathcal{B}(\ell) = \textsf{False}}{value(\mathcal{B}; \neg R/\ell) = R \mapsto \texttt{true}}\,(\text{VAL}-\neg\text{T}-\text{TRUE})$$

Figure B.14: Using $\mathcal{B}$ to get the value of an effect N

B.6 Functions

The semantics will use 6 functions that use set creation to produce new substitutions and lattices.

The first two functions, seen in Figure B.15 are for creating sets of substitutions. The function findLabels will, given a lattice $\mathcal{A}$, a binding β, and specification types as in Γ_y, return the set of all substitutions possible from $\mathcal{A}$ and β such that each substitution has the domain given in β and respects the types given in Γ_y. The domain of Γ_y may be larger than the domain of β. The second function, allValidSubs, does something similar, but it is not limited by β. Instead, it will create all substitutions based upon the entire domain of Γ_y such that the types of Γ_y are respected and that each substitution created is a superset of the given substitution σ. That is, it will use σ as a starting point for creating further substitutions based on $\Gamma_v ary$.

The next 3 functions, seen in Figure B.17, will generate effect lattices δ. The functions ignore and $\bot$ are straightforward and will create a δ such that every R is mapped to $*$ and bot respectively. The function lattice will create a delta lattice from the effects list of a constraint. It will do so given a specific substitution σ and a $\mathcal{B}$ to use for test effects. Notice that when multiple effects are made, they can override each other such that later effects override earlier effects.

The last function, transfer, in Figure B.18, will transfer a relationship lattice into a new domain, as dictated by $\mathcal{A}$. As the flow analysis proceeds, the lattice $\mathcal{A}$ will gain new variables x and object labels ℓ. These new object labels will cause new relationships to be possible. The function transfer adds these new relationships and sets them to the default starting value, Unknown.

$\boxed{\text{findLabels}(< \Gamma_\ell; \mathcal{L} >; \beta; \Gamma_y) = \Sigma}$

$$\Sigma = \{\sigma' \mid \sigma = \{y \mapsto \ell \mid y \in \text{dom}(\beta) \ \wedge \ \ell \in \mathcal{L}(\beta(y)) \ \wedge \ \exists \tau' . \tau' <: \Gamma_\ell(\ell) \ \wedge \ \tau' <: \Gamma_y(y)\} \ \wedge$$

$$\sigma' \in \text{allValidSubs}(< \Gamma_\ell; \mathcal{L} >; \sigma; \Gamma_y)\}$$

$\boxed{\text{allValidSubs}(< \Gamma_\ell; \mathcal{L} >; \sigma; \Gamma_y) = \Sigma}$

$$\Sigma = \{\sigma' \mid \sigma' \supseteq \sigma \ \wedge \ \text{dom}(\sigma') = \text{dom}(\Gamma_y) \ \wedge \ \forall y \mapsto \ell \in \sigma' . \exists \tau' . \tau' <: \Gamma_\ell(\ell) \ \wedge \ \tau' <: \Gamma_y(y)\}$$

Figure B.15: Functions to create substitutions

$$\boxed{\perp(\sigma) = \alpha}$$

$$\alpha = \{y \mapsto \varnothing \mid y \in \mathrm{dom}(\sigma)\}$$

Figure B.16: Creating an empty update

$$\boxed{\mathsf{ignore}(< \Gamma_\ell; \mathcal{L} >) = \delta}$$

$$\delta = \{\mathsf{rel}(\bar{\ell}) \mapsto * \mid \mathcal{R}(\mathsf{rel}) = \bar{\tau} \wedge |\bar{\tau}| = |\bar{\ell}| = n \wedge \forall_{i=1}^n . \exists \tau' . \tau' <: \tau_i \wedge \tau' <: \Gamma_\ell(\ell_i)\}$$

$$\boxed{\perp(< \Gamma_\ell; \mathcal{L} >) = \delta}$$

$$\delta = \{\mathsf{rel}(\bar{\ell}) \mapsto \mathtt{bot} \mid \mathcal{R}(\mathsf{rel}) = \bar{\tau} \wedge |\bar{\tau}| = |\bar{\ell}| = n \wedge \forall_{i=1}^n . \exists \tau' . \tau' <: \tau_i \wedge \tau' <: \Gamma_\ell(\ell_i)\}$$

$$\boxed{\mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; Q, \bar{Q}) = \delta}$$

$$\delta = \mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; Q) \sqcap \mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q})$$

$$\boxed{\mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; Q) = \delta}$$

$$\delta = \mathsf{ignore}(\mathcal{A}) \sqcap \{value(\mathcal{B}; Q[\sigma']) \mid \sigma' \in \mathsf{allValidSubs}(\mathcal{A}; \sigma; FV(Q))\}$$

$$\boxed{\mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \varnothing) = \delta}$$

$$\delta = \mathsf{ignore}(\mathcal{A})$$

Figure B.17: Functions to create an effect lattice δ.

$$\boxed{\mathsf{transfer}(\rho; \mathcal{A}) = \rho'}$$

$$\rho' = \{R \mapsto t \mid R = \mathsf{rel}(\bar{\ell}) \wedge \mathcal{R}(\mathsf{rel}) = \bar{\tau} \wedge |\bar{\tau}| = |\bar{\ell}| = n \wedge \forall_{i=1}^n . \exists \tau' . \tau' <: \tau_i \wedge \tau' <:$$

$$\Gamma_\ell(\ell_i) \wedge (R \in \mathrm{dom}(\rho) \implies t = \rho(R)) \wedge (R \notin \mathrm{dom}(\rho) \implies t = \mathsf{Unknown})\}$$

Figure B.18: Transfer lattice into new aliasing domain function

B.7   Rules

This section will describe the formal rules for the flow function by starting with the lowest level rules and working back up.

At the core of the analysis is a simple logic engine. This logic engine will simply evaluate whether a given relationship predicate, M is satisfied by the context ρ. The rules for this are shown in Figures B.19-B.22.

While most of the rules (Figures B.21 and B.22) are as one would expect for a three-value logic system and the same for all variants, Figure B.19 shows an interesting difference. In the sound and complete variants, the rule for checking the atomic relationship R is a trivial lookup into ρ (REL). This is also the case in the pragmatic variant when the relationship maps to either True or False (REL-T-F). The interesting case is in the pragmatic variant when the relationship maps to Unknown. The pragmatic variant admits the rules (REL-U) and (INFER) to handle this case. These rules attempt to use the inferred relationships, defined in Section 4.3.4, to retrieve the desired relationship.

The rule for the inference judgement ρ infers ρ' is defined in Figure B.20. This rule first checks to see if the trigger of an inferred relation is true, and if so, uses the function $lattice$ to produce the inferred relationships described by $\bar{R}[\sigma]$. For all relationships not defined by $\bar{R}[\sigma]$, $lattice$ defaults to bot to signal that there are no changes. There are two properties to note about the rules (REL-U), (INFER), and (DISCOVER):

1. The use of inferred relationships does not change the original lattice ρ. This allows the inferred relationships to disappear if the generator, P, is no longer true.

2. Any inferred values must be *strictly more precise* than the relationship's value in ρ, as enforced by $\rho' \sqsubseteq \rho$. This means that relationships can move from Unknown to True, but they can not move from False to True. This property guarantees termination and gives declared effects precedence over inferred ones.

Inferred relationships can not be used in the sound and complete variants. This does not limit the expressiveness of the specifications, as inferred relations can always be written directly within the constraints. Doing so does make the specifications more difficult to write; the framework developer must add the inferred relations to any constraint which will also prove the trigger predicate. Since inferred relations do change the semantics, they are not syntactic sugar, but they are not necessary for reasons beyond the ease of writing specifications.

$\boxed{\mathcal{A};\mathcal{B};\rho \vdash R\ t}$ Sound and Complete variants

$$\frac{\rho(R) = t}{\mathcal{A};\mathcal{B};\rho \vdash R\ t}\text{(REL)}$$

$\boxed{\mathcal{A};\mathcal{B};\rho \vdash R\ t}$ Pragmatic variants

$$\frac{\rho(R) = t \qquad t \neq \text{Unknown}}{\mathcal{A};\mathcal{B};\rho \vdash R\ t}\text{(REL−T−F)}$$

$$\frac{\rho(R) = \text{Unknown} \qquad \mathcal{A};\mathcal{B} \vdash \rho \text{ infers } \rho' \qquad \rho \leftleftarrows \rho' \vdash R\ t \qquad t \neq \text{Unknown}}{\mathcal{A};\mathcal{B};\rho \vdash R\ t}\text{(INFER)}$$

$$\frac{\rho(R) = \text{Unknown}}{\neg\exists\rho\ .\ \mathcal{A};\mathcal{B} \vdash \rho \text{ infers } \rho' \ \wedge\ \rho \leftleftarrows \rho' \vdash R\ t\ \wedge\ t \neq \text{Unknown}}{\mathcal{A};\mathcal{B};\rho \vdash R\ \text{Unknown}}\text{(REL−U)}$$

Figure B.19: Three value truth evaluation on M, continued on B.21. The sound and complete variant use only the rule $\mathrm{rel - sound - complete}$, the other rules are for the pragmatic variant.

$\boxed{\mathcal{A};\mathcal{B} \vdash \rho \text{ infers } \rho'}$

$$\frac{P \Downarrow \bar{Q} \in \mathcal{I} \qquad \mathcal{A};\mathcal{B};\rho \vdash P[\sigma]\ \text{True} \qquad \text{lattice}(\mathcal{A};\mathcal{B};\sigma;\bar{Q}) = \delta \qquad \rho' = \rho \Leftarrow \delta \qquad \rho' \sqsubset \rho}{\mathcal{A};\mathcal{B} \vdash \rho \text{ infers } \rho'}\text{(DISCOVER)}$$

Figure B.20: Inferred Relationship Discovery.

$\boxed{\mathcal{A}; \mathcal{B}; \rho \vdash M\ t}$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash R\ t \qquad \mathcal{B}(\ell_{\text{test}}) = t \qquad t \neq \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash R/\ell_{\text{test}}\ \text{True}}\ (\text{REL}-\text{TEST}-\text{T})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash R\ t_1 \qquad \mathcal{B}(\ell_{\text{test}}) = t_2 \qquad t_1 \neq \text{Unknown} \qquad t_2 \neq \text{Unknown} \qquad t_1 \neq t_2}{\mathcal{A}; \mathcal{B}; \rho \vdash R/\ell_{\text{test}}\ \text{False}}\ (\text{REL}-\text{TEST}-\text{F})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash R\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash R/\ell_{\text{test}}\ \text{Unknown}}\ (\text{REL}-\text{TEST}-\text{U1}) \qquad \frac{\mathcal{B}(\ell_{\text{test}}) = \text{Unknown} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash R\ t}{\mathcal{A}; \mathcal{B}; \rho \vdash R/\ell_{\text{test}}\ \text{Unknown}}\ (\text{REL}-\text{TEST}-\text{U2})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash T\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash \neg T\ \text{Unknown}}\ (\neg\text{T}-\text{U}) \qquad \frac{\mathcal{A}; \mathcal{B}; \rho \vdash T\ \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash \neg T\ \text{True}}\ (\neg\text{T}-\text{T}) \qquad \frac{\mathcal{A}; \mathcal{B}; \rho \vdash T\ \text{True}}{\mathcal{A}; \mathcal{B}; \rho \vdash \neg T\ \text{False}}\ (\neg\text{T}-\text{F})$$

$$\frac{}{\mathcal{A}; \mathcal{B}; \rho \vdash true\ \text{True}}\ (\text{TRUE}) \qquad\qquad \frac{}{\mathcal{A}; \mathcal{B}; \rho \vdash false\ \text{False}}\ (\text{FALSE})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \implies M_2\ \text{True}}\ (\implies -\text{T1}) \qquad \frac{\mathcal{A}; \mathcal{B}; \rho \vdash P_2\ \text{True}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \implies M_2\ \text{True}}\ (\implies -\text{T2})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{True} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \implies M_2\ \text{False}}\ (\implies -\text{F})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{Unknown} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \implies M_2\ \text{Unknown}}\ (\implies -\text{U1})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{True} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \implies M_2\ \text{Unknown}}\ (\implies -\text{U2})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{Unknown} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \implies M_2\ \text{Unknown}}\ (\implies -\text{U3})$$

Figure B.21: Three value truth evaluation on M, continued on B.22.

$$\mathcal{A}; \mathcal{B}; \rho \vdash M\ t$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{True} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{True}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \wedge M_2\ \text{True}}(\wedge-\text{T}) \qquad \frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \wedge M_2 \text{False}}(\wedge-\text{F1})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \wedge M_2\ \text{False}}(\wedge-\text{F2}) \qquad \frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{True} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \wedge M_2\ \text{Unknown}}(\wedge-\text{U1})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{Unknown} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{True}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \wedge M_2\ \text{Unknown}}(\wedge-\text{U2})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{Unknown} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \wedge M_2\ \text{Unknown}}(\wedge-\text{U3})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{True}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \vee M_2\ \text{True}}(\vee-\text{T1}) \qquad \frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{True}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \vee M_2\ \text{True}}(\vee-\text{T2})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{False} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \vee M_2\ \text{False}}(\vee-\text{F})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{False} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \vee M_2\ \text{Unknown}}(\vee-\text{U1})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{Unknown} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{False}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \vee M_2\ \text{Unknown}}(\vee-\text{U2})$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash M_1\ \text{Unknown} \qquad \mathcal{A}; \mathcal{B}; \rho \vdash M_2\ \text{Unknown}}{\mathcal{A}; \mathcal{B}; \rho \vdash M_1 \vee M_2\ \text{Unknown}}(\vee-\text{U3})$$

Figure B.22: Three value truth evaluation on M, continued from B.21.

$\boxed{\text{instr} : \text{op} \Mapsto \beta}$

$$\frac{\begin{array}{c} \exists \tau' . \, \tau' <: \tau_{\text{this}} \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x}_{\text{this}}) \\ \exists \tau' . \, \tau' <: \tau_{\text{ret}} \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x}_{\text{ret}}) \qquad \overline{\exists \tau' . \, \tau' <: \tau \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x})} \end{array}}{\mathbf{x}_{\text{ret}} = \mathbf{x}_{\text{this}}.\mathsf{m}(\overline{\mathbf{x}}) : \tau_{\text{this}}.\mathsf{m}(\overline{\tau}\,\overline{\mathsf{y}}) : \tau_{\text{ret}} \Mapsto \mathbf{x}_{\text{ret}} \mapsto \mathsf{result}, \mathbf{x}_{\text{this}} \mapsto \mathsf{target}, \overline{\mathbf{x} \mapsto \mathsf{y}}} (\text{INVOKE})$$

$$\frac{\exists \tau' . \, \tau' <: \mathsf{new} \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x}_{\text{ret}}) \qquad \overline{\exists \tau' . \, \tau' <: \tau \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x})}}{\mathbf{x}_{\text{ret}} = \mathsf{new} \, \mathsf{m}(\overline{\mathbf{x}}) : \mathsf{new} \, \tau(\overline{\tau}\,\overline{\mathsf{y}}) \Mapsto \mathbf{x}_{\text{ret}} \mapsto \mathsf{target}, \overline{\mathbf{x} \mapsto \mathsf{y}}} (\text{CONSTRUCTOR})$$

$$\frac{\begin{array}{c} \exists \tau' . \, \tau' <: \tau_{\text{this}} \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x}_{\text{this}}) \\ \exists \tau' . \, \tau' <: \tau_{\text{ret}} \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x}_{\text{ret}}) \qquad \overline{\exists \tau' . \, \tau' <: \tau \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x})} \end{array}}{\mathtt{return}\, \mathbf{x}_{\text{ret}}(\mathbf{x}_{\text{this}}.\mathsf{m}(\overline{\mathbf{x}})) : \mathsf{eom}(\tau_{\text{this}}.\mathsf{m}(\overline{\tau}\,\overline{\mathsf{y}}) : \tau_{\text{ret}}) \Mapsto \mathbf{x}_{\text{ret}} \mapsto \mathsf{result}, \mathbf{x}_{\text{this}} \mapsto \mathsf{target}, \overline{\mathbf{x} \mapsto \mathsf{y}}} (\text{EOM})$$

$$\frac{\exists \tau' . \, \tau' <: \tau_{\text{this}} \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x}_{\text{this}}) \qquad \overline{\exists \tau' . \, \tau' <: \tau \, \wedge \, \tau' <: \Gamma_{\mathbf{x}}(\mathbf{x})}}{\mathtt{begin}(\mathbf{x}_{\text{this}}.\mathsf{m}(\overline{\mathbf{x}})) : \mathsf{bom}(\tau_{\text{this}}.\mathsf{m}(\overline{\tau}\,\overline{\mathsf{y}})) \Mapsto \mathbf{x}_{\text{this}} \mapsto \mathsf{target}, \overline{\mathbf{x} \mapsto \mathsf{y}}} (\text{BOM})$$

Figure B.23: Instruction binding.

In order to check a constraint, the analysis must determine whether a source instruction, called `instr`, matches the operation op defined by a constraint, and it must bind up source variables x to specification variables y, as contained in β. The rules for are defined in Figure B.23. The rules match variables appropriately and ensure that there exists some typing possibility that would make them compatible. These rules can be expanded to allow for new types of operations.

$$\boxed{\mathcal{A}; \mathcal{B}; \rho; \text{cons} \vdash \text{instr} \hookrightarrow \delta, \gamma} \; \text{Assume } \text{cons} = \text{op} : P_{\text{ctx}} \Rightarrow P_{\text{req}} \Downarrow \bar{Q}; P_{\text{rst}}$$

$$\frac{\begin{array}{c} \text{instr} : \text{op} \mapsto \beta \quad \Gamma_y = FV(\text{op}) \cup FV(P_{\text{ctx}}) \cup FV(\bar{Q}) \quad \text{findLabels}(\mathcal{A}; \Gamma_y; \beta) = \Sigma \\ \Sigma \neq \varnothing \quad \mathcal{T} = \{\sigma, \delta, \gamma \mid \sigma \in \Sigma \wedge \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha \wedge \gamma = \alpha[\beta]\} \\ \mathcal{T}.\sigma = \Sigma \quad \delta' = \sqcup \mathcal{T}.\delta \quad \gamma' = \sqcup \mathcal{T}.\gamma \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \text{cons} \vdash \text{instr} \hookrightarrow \delta', \gamma'} \; (\text{MATCH})$$

$$\frac{\text{instr} : \text{op} \mapsto \beta \quad \Gamma_y = FV(\text{op}) \cup FV(P_{\text{ctx}}) \cup FV(\bar{Q}) \quad \text{findLabels}(\mathcal{A}; \Gamma_y; \beta) = \varnothing}{\mathcal{A}; \mathcal{B}; \rho; \text{cons} \vdash \text{instr} \hookrightarrow \bot(\mathcal{A}), \varnothing} \; (\text{NO\!-\!ALIASES})$$

$$\frac{\neg(\text{instr} : \text{op} \mapsto \beta)}{\mathcal{A}; \mathcal{B}; \rho; \text{cons} \vdash \text{instr} \hookrightarrow \bot(\mathcal{A}), \varnothing} \; (\text{NO\!-\!MATCH})$$

Figure B.24: Check a single constraint on all possible alias bindings given the two alias environments.

With these pieces in place, I will now show how to check a single constraint. This is done with the judgment

$$\mathcal{A}; \mathcal{B}; \rho; \text{cons} \vdash \text{instr} \hookrightarrow \delta, \gamma$$

shown in Figure B.24. This judgment takes the environments and a constraint, and it determines how to change the lattices for a given instruction. The lattice changes are represented in δ, and the alias changes are represented in γ.

The analysis starts by checking whether the instruction matches the constrained operation using the instruction matching rules from Figure B.23. If not, the rule (NO-MATCH) will apply. If there is a match, it will also check whether the binding provided can produce any substitutions. If no substitutions are available, then rule (NO-ALIASES) applies. Both of these rules produce no lattice effects.

If there are substitutions, as shown in rule (MATCH), then the analysis must check this constraint for every aliasing configuration possible, as represented by Σ. This rule checks that for each substitution σ, the constraint passes and produces the change lattices δ and α. The α for each substitution is converted into a γ using the bindings for the instruction. Once the analysis has all change lattices for each substitution, the analysis combines them together using the $\sqcup$ operator and returns the combined change lattices.

As seen in Figure B.24, the rule (MATCH) must check the validity of each possible substitution. This is done with the judgment

$$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha$$

The rules for this judgment, shown in Figure B.25, are the primary point of difference between the variants of the analysis. The differences are highlighted for convenience. The rules for this judgment will all use the function lattice to produce the relationship delta lattice when appropriate, and they will use the restriction rules in Figure B.26 to produce the alias delta lattice.

Sound Variant. The sound variant first checks $P_{trg}[\sigma]$ under ρ. It uses this to determine which rule applies. If $P_{trg}[\sigma]$ is True, as seen in rule (BOUND-T), then the analysis must check if P_{req} is True under ρ for all substitutions. If P_{req} is not True with all substitution from Σ, then the analysis produces an error. If there is no error, the rule produces the effects dictated by the function $lattice$ and will produce effects based upon the restriction judgment. If $P_{trg}[\sigma]$ is False, then the analysis uses rule (BOUND-F). In this situation the constraint does not trigger, so the requires predicate is not checked. The analysis returns no delta lattice changes, and it returns σ so that this substitution is not restricted.

In the case that $P_{trg}[\sigma]$ is Unknown, the sound variant proceeds in a similar manner to the case where $P_{trg}[\sigma]$ is True as it must consider the possibility that the trigger predicate is actually true, as seen in (BOUND-U) The only difference is in how it treats effects. The analysis must use the polarizing operator to be conservative with the effects it is producing in case the trigger predicate is actually false at runtime. Likewise, it will always produce the aliasing change effect σ.

Complete Variant. Like the sound variant, the complete variant starts by checking $P_{trg}[\sigma]$ under ρ. If $P_{trg}[\sigma]$ is True, as seen in rule (BOUND-T), then the analysis must check P_{req} under ρ given any substitution. As this is the complete variant, the analysis does not care whether P_{req} is True or Unknown. If no substitutions work, either because none exist or because they all show P_{req} to be false, then the analysis produces an error. Otherwise, the rule produces effects as expected. If the analysis determines that $P_{trg}[\sigma]$ is False, then it uses the rule (BOUND-F). Like the sound variant, the requires predicate is not checked, the analysis returns no delta lattice changes, and it returns σ so that this substitution is not restricted.

Finally, if $P_{trg}[\sigma]$ is Unknown, the complete variant will not check P_{req} as it cannot be sure whether the constraint is actually triggered and it should not produce an error. However, it must still produce some conservative effects in case the constraint is triggered given a more concrete lattice. Like the sound rule in the case of an unknown trigger, the rule uses the polarizing operator $\overset{\uparrow}{*}$ to produce only conservative effects, and it produces the aliasing change effect σ.

Pragmatic Variant. The pragmatic variant is a combination of the sound and complete variants. It has the same rule for False as the other two variants, (BOUND-F). The rule (BOUND-U) for pragmatic is also the same as the rule (BOUND-U) for completeness. This means that this variant can produce both false positives and false negatives. False negatives can occur when P_{trg} is Unknown under ρ, but a more precise lattice would have found P_{trg} to be True and eventually generated an error. False positives occur when P_{trg} is True under ρ and P_{req} is Unknown under ρ, but P_{req} would have been True under a more precise lattice.

For all of these rules, $\text{cons} = \text{op} : P_{\text{ctx}} \Rightarrow P_{\text{req}} \Downarrow \bar{Q}; P_{\text{rst}}$

$\boxed{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha \text{ (Pragmatic)}}$

$$\frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ True} \\ \text{allValidSubs}(\mathcal{A}; \sigma; \text{FV}(\text{cons})) = \Sigma \quad \boxed{\exists\, \sigma' \in \Sigma \,.\, \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{req}}[\sigma'] \text{ True}} \\ \text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \quad \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_{\alpha} P_{\text{rst}} \hookrightarrow \alpha \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha} \text{(BOUND--T)}$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ False}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \bot(\mathcal{A}), \sigma} \text{(BOUND--F)} \qquad \frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ Unknown} \\ \text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow^{\uparrow}_{*} \delta, \sigma} \text{(BOUND--U)}$$

$\boxed{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha \text{ (Sound)}}$

$$\frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ True} \\ \text{allValidSubs}(\mathcal{A}; \sigma; \text{FV}(\text{cons})) = \Sigma \quad \boxed{\forall\, \sigma' \in \Sigma \,.\, \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{req}}[\sigma'] \text{ True}} \\ \text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \quad \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_{\alpha} P_{\text{rst}} \hookrightarrow \alpha \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha} \text{(BOUND--T)}$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ False}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \bot(\mathcal{A}), \sigma} \text{(BOUND--F)}$$

$$\frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ Unknown} \\ \text{allValidSubs}(\mathcal{A}; \sigma; \text{FV}(\text{cons})) = \Sigma \quad \boxed{\forall\, \sigma' \in \Sigma \,.\, \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{req}}[\sigma'] \text{ True}} \\ \text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow^{\uparrow}_{*} \delta, \sigma} \text{(BOUND--U)}$$

$\boxed{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha \text{ (Complete)}}$

$$\frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ True} \quad \text{allValidSubs}(\mathcal{A}; \sigma; \text{FV}(\text{cons})) = \Sigma \\ \boxed{\exists\, \sigma' \in \Sigma \,.\, \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{req}}[\sigma'] \text{ True} \vee \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{req}}[\sigma'] \text{ Unknown}} \\ \text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \quad \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_{\alpha} P_{\text{rst}} \hookrightarrow \alpha \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \delta, \alpha} \text{(BOUND--T)}$$

$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ False}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow \bot(\mathcal{A}), \sigma} \text{(BOUND--F)} \qquad \frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \text{ Unknown} \\ \text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \hookrightarrow^{\uparrow}_{*} \delta, \sigma} \text{(BOUND--U)}$$

Figure B.25: Check a bound constraint.

$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \hookrightarrow \alpha$ (Sound and Complete)

$$\frac{\begin{array}{c} \Sigma = \mathsf{allValidSubs}(\mathcal{A}; \sigma, \mathsf{FV}(P)) \\ \exists \sigma' \in \Sigma. \; \mathcal{A}; \mathcal{B}; \rho \vdash P[\sigma'] \; t \qquad t \neq \mathsf{False} \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \hookrightarrow \sigma} \; (\text{RESTRICT}-\text{T}-\text{U}-\text{SOUND}/\text{COMPLETE})$$

$$\frac{\begin{array}{c} \Sigma = \mathsf{allValidSubs}(\mathcal{A}; \sigma, \mathsf{FV}(P)) \\ \forall \sigma' \in \Sigma. \; \mathcal{A}; \mathcal{B}; \rho \vdash P[\sigma'] \; \mathsf{False} \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \hookrightarrow \bot(\sigma)} \; (\text{RESTRICT}-\text{F}-\text{SOUND}/\text{COMPLETE})$$

$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \hookrightarrow \alpha$ (Pragmatic)

$$\frac{\begin{array}{c} \Sigma = \mathsf{allValidSubs}(\mathcal{A}; \sigma, \mathsf{FV}(P)) \\ \exists \sigma' \in \Sigma. \; \mathcal{A}; \mathcal{B}; \rho \vdash P[\sigma'] \; \mathsf{True} \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \hookrightarrow \sigma} \; (\text{RESTRICT}-\text{T}-\text{PRAGMATIC})$$

$$\frac{\begin{array}{c} \Sigma = \mathsf{allValidSubs}(\mathcal{A}; \sigma, \mathsf{FV}(P)) \\ \forall \sigma' \in \Sigma. \; \mathcal{A}; \mathcal{B}; \rho \vdash P[\sigma'] \; t \; \wedge \; t \neq \mathsf{True} \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \hookrightarrow \bot(\sigma)} \; (\text{RESTRICT}-\text{F}-\text{U}-\text{PRAGMATIC})$$

Figure B.26: Restricting substitutions based on a predicate.

When the analysis is checking a constraint, it may find a restrict predicate and need to restrict the aliases of a variable accordingly. This is done in the rule (BOUND-T) in Figure B.25. The rules in Figure B.26 show how, given a predicate, the analysis determines which aliases to restrict. The substitutions to restrict to are returned from the rule with the lattice α. As before, the pragmatic variant works different from the sound and complete variants, as shown by the shading. The sound and complete variants will only restrict a substitution σ if there are no possible ways to make the predicate True or False, as seen in rule (RESTRICT-F-SOUND/COMPLETE). If there is any way for the substitution to make the predicate True or Unknown, it will return σ as a potentially valid substitution. This is the only way to safely restrict substitutions, but as Unknown is a fairly common result, it means that restriction happens only in rare circumstances when the analysis has very precise knowledge.

The pragmatic variant attempts to rectify this by allowing for unsafe restrictions. In particular, it treats Unknown the same way it treats False, as seen in rules (RESTRICT-T-PRAGMATIC) and (RESTRICT-F-U-PRAGMATIC). This allows for more aggressive restrictions, which in practice are usually acceptable.

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

$$\frac{\begin{array}{cc} f_{alias}(\mathcal{A}, \text{instr}) = \mathcal{A}' & \mathcal{T} = \{\text{cons}, \delta, \gamma \mid \text{cons} \in \mathcal{C} \wedge \mathcal{A}'; \mathcal{B}; \rho; \text{cons} \vdash \text{instr} \hookrightarrow \delta, \gamma\} \\ \mathcal{T}.\text{cons} = \mathcal{C} & \mathcal{A}' \Leftarrow (\sqcup \mathcal{T}.\gamma) = \mathcal{A}'' \quad \text{transfer}(\rho, \mathcal{A}'') \Leftarrow (\boxminus \mathcal{T}.\delta) = \rho' \end{array}}{f_{\mathcal{C}}(\mathcal{A}; \mathcal{B}; \rho; \text{instr}) = \rho', \mathcal{A}''} (\text{FLOW–CONS})$$

Figure B.27: Flow function

Finally, I present the semantics for the flow function of the analysis in Figure B.27. The flow function for the Fusion analysis checks all the individual constraints and produces the output lattices for the instruction. The flow function starts by first calling the alias analysis to produce the new alias lattice for the instruction. Then, using the judgments defined previously, the flow function iterates through each constraint and receives the change lattices δ and γ. The γ lattices are all combined using the $\sqcup$ operator, and the changes are applies to the incoming alias lattice $\mathcal{A}'$ to produce the outgoing alias lattice $\mathcal{A}''$.

The δ lattices are combined as well, but we use the $\boxminus$ operator here instead. This operator will effectively remove the $\texttt{true}*$ and $\texttt{false}*$ elements from the lattice. This operator will allow $\texttt{true}*$ to be effectively changed to $\texttt{true}$ as long as all the substitutions agreed to it and at least one substitution definitely made the change to $\texttt{true}$; this preserves some precision even in cases where there are a lot of Unknown predicates as long as at least one made a concrete change. Once the analysis has the final change lattice δ, it transfers the lattice ρ into the new aliasing context $\mathcal{A}''$ and applies the effects.

There are three final rules that are not used in the semantics above but are necessary for the proofs in Appendix C, these are shown in Figure B.28. The first and second show that there is a consistency between an $\mathcal{A}$ and a ρ or δ such that all labels in ρ or δ exist in $\mathcal{A}$ with the right type and that the domain of ρ of δ contains all possible relationships that can be created under $\mathcal{A}$. The second shows the consistency between an $\mathcal{A}$, a σ, and a Γ_y. This shows that under some $\mathcal{A}$, σ contains a valid substitution for every y in Γ_y.

$\boxed{\mathcal{A} \vdash \rho \text{ consistent}}$

$$\dfrac{\text{dom}(\rho) = \{\text{rel}(\bar{\ell}) \mid \bar{\tau} = \mathcal{R}(\text{rel}) \ \wedge \ |\bar{\tau}| = |\bar{\ell}| = n \ \wedge \ \forall_{i=1}^{n} . \ \exists \tau' . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell(\ell_i)\}}{< \Gamma_\ell; \mathcal{L} > \vdash \rho \text{ consistent}} \text{(CONSISTENT}-\rho)$$

$\boxed{\mathcal{A} \vdash \delta \text{ consistent}}$

$$\dfrac{\text{dom}(\delta) = \{\text{rel}(\bar{\ell}) \mid \bar{\tau} = \mathcal{R}(\text{rel}) \ \wedge \ |\bar{\tau}| = |\bar{\ell}| = n \ \wedge \ \forall_{i=1}^{n} . \ \exists \tau' . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell(\ell_i)\}}{< \Gamma_\ell; \mathcal{L} > \vdash \delta \text{ consistent}} \text{(CONSISTENT}-\delta)$$

$\boxed{\mathcal{A} \vdash \sigma \text{ validFor } \Gamma_y}$

$$\dfrac{\text{dom}(\sigma) \supseteq \text{dom}(\Gamma_y) \qquad \forall \, y : \tau \in \Gamma_y . \ \exists \tau' . \ \tau' <: \Gamma_\ell(\sigma(y)) \ \wedge \ \tau' <: \tau}{< \Gamma_\ell; \mathcal{L} > \vdash \sigma \text{ validFor } \Gamma_y} \text{(}\sigma-\text{VALID)}$$

Figure B.28: Consistency of ρ and validity of σ against $\mathcal{A}$

C

Appendix

Proofs of Soundness and Completeness

C.1 Soundness

Global soundness from local soundness, consistency, monotonicity, and sound aliasing and sound boolean propagation.

Theorem 1. Soundness of Flow Function

$$\mathit{forall\ der.}$$

$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$

$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$

$$\mathcal{A}^{abs} \vdash \rho^{abs} \text{ consistent}$$

$$\mathcal{A}^{conc} \vdash \rho^{conc} \text{ consistent}$$

$$\rho^{conc} \sqsubseteq \rho^{abs}$$

$$f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs}; \mathrm{instr}) = \rho^{abs\prime}, \mathcal{A}^{abs\prime\prime}$$

$$\mathit{exists\ der.}$$

$$f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; \mathrm{instr}) = \rho^{conc\prime}, \mathcal{A}^{conc\prime\prime}$$

$$\rho^{conc\prime} \sqsubseteq \rho^{abs\prime}$$

$$\mathcal{A}^{conc\prime\prime} \sqsubseteq \mathcal{A}^{abs\prime\prime}$$

Proof:

$\mathcal{T}^{abs} = \{\mathrm{cons}, \delta, \gamma \mid \mathrm{cons} \in \mathcal{C} \wedge \mathcal{A}^{abs\prime}; \mathcal{B}^{abs}; \rho^{abs}; \mathrm{cons} \vdash \mathrm{instr} \hookrightarrow \delta, \gamma\}$

By inversion on $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs}; \mathrm{instr}) = \rho^{abs\prime}, \mathcal{A}^{abs\prime\prime}$

$\mathcal{T}^{abs}.\mathrm{cons} = \mathcal{C}$ By inversion on $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs}; \mathrm{instr}) = \rho^{abs\prime}, \mathcal{A}^{abs\prime\prime}$

$\gamma^{abs} = \sqcup \mathcal{T}^{abs}.\gamma$ By inversion on $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs}; \mathrm{instr}) = \rho^{abs\prime}, \mathcal{A}^{abs\prime\prime}$

$\mathcal{A}^{abs\prime\prime} = \mathcal{A}^{abs\prime} \Leftarrow \gamma^{abs}$ By inversion on $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs}; \mathrm{instr}) = \rho^{abs\prime}, \mathcal{A}^{abs\prime\prime}$

157

$\delta^{abs} = \boxminus \mathcal{T}^{abs}.\delta$ $\qquad$ By inversion on $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs};instr) = \rho^{abs'}, \mathcal{A}^{abs''}$

$\rho^{abs'} = \mathsf{transfer}(\rho^{abs}, \mathcal{A}^{abs''}) \Leftarrow \delta^{abs}$ $\qquad$ By inversion on $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs};instr) = \rho^{abs'}, \mathcal{A}^{abs''}$

$f_{alias}(\mathcal{A}^{abs};instr) = \mathcal{A}^{abs'}$ $\qquad$ By inversion on $f_{\mathcal{C};\mathcal{A}^{abs};\mathcal{B}^{abs}}(\rho^{abs};instr) = \rho^{abs'}, \mathcal{A}^{abs''}$

$f_{alias}(\mathcal{A}^{conc};instr) = \mathcal{A}^{conc'}$ $\qquad\qquad\qquad$ By Theorem Aliasing Flow Function Sound

$\mathcal{A}^{conc'} \sqsubseteq_{\mathcal{A}} \mathcal{A}^{abs'}$ $\qquad\qquad\qquad\qquad$ By Theorem Aliasing Flow Function Sound

$\mathcal{A}^{conc'} \vdash \rho^{conc'}$ consistent $\qquad\qquad$ By Theorem Aliasing Flow Function Preserves Consistency

$\forall\, cons \in \mathcal{C}\,.$

$\quad$ Let $\mathcal{A}^{abs'};\ \mathcal{B}^{abs};\ \rho^{abs};\ cons \vdash instr \hookrightarrow \delta^{a}, \gamma^{a}$ $\qquad\qquad$ By construction of $\mathcal{T}^{abs}$

$\quad\mathcal{A}^{conc'};\ \mathcal{B}^{conc};\ \rho^{conc};\ cons \vdash instr \hookrightarrow \delta^{c}, \gamma^{c}$ $\qquad$ By Lemma 1(Soundness of Single Constraint)

$\quad\ \delta^{c} \sqsubseteq \delta^{a}$ $\qquad\qquad\qquad\qquad\qquad$ By Lemma 1(Soundness of Single Constraint)

$\quad\ \gamma^{c} \sqsubseteq \gamma^{a}$ $\qquad\qquad\qquad\qquad\qquad$ By Lemma 1(Soundness of Single Constraint)

$\quad\ \mathcal{A}^{conc'} \vdash \delta^{c}$ consistent $\qquad\qquad\qquad$ By Lemma 9(Consistency of a Single Constraint)

$\quad\ \mathsf{dom}(\gamma^{c}) \subseteq \mathsf{dom}(\mathcal{A}^{conc'}.\mathcal{L})$ $\qquad\qquad$ By Lemma 9(Consistency of a Single Constraint)

Let $\mathcal{T}^{conc} = \{cons, \delta, \gamma \mid cons \in \mathcal{C} \land \mathcal{A}^{conc'};\ \mathcal{B}^{conc};\ \rho^{conc};\ cons \vdash instr \hookrightarrow \delta, \gamma\}$

$\mathcal{T}^{conc}.cons = \mathcal{C}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By set construction

Let $\delta^{conc} = \boxminus \mathcal{T}^{conc}.\delta$ $\qquad\qquad\qquad\qquad\qquad\qquad$ By rule (EQJOIN)

$\delta^{conc} \sqsubseteq \delta^{abs}$ $\qquad\qquad\qquad$ By Lemma 30(eqjoin delta operator preserves alap)

Let $\gamma^{conc} = \sqcup \mathcal{T}^{conc}.\gamma$ $\qquad\qquad\qquad\qquad\qquad\qquad$ By rule $(\sqcup_{\delta})$

$\gamma^{conc} \sqsubseteq \gamma^{abs}$ $\qquad\qquad\qquad$ By Lemma 33($\sqcup_{\gamma}$ operator preserves $\sqsubseteq$)

Let $\mathcal{A}^{conc''} = \mathcal{A}^{conc'} \Leftarrow \gamma$ $\qquad\qquad\qquad\qquad\qquad$ By rule $(\Leftarrow_{\mathcal{A}})$

$\mathcal{A}^{conc''} \sqsubseteq \mathcal{A}^{abs''}$ $\qquad\qquad\qquad$ By Lemma 34(Leftarrow AliasA preserves alap)

Let $\rho^{conc''} = \mathsf{transfer}(\rho^{conc}, \mathcal{A}^{conc''})$ $\qquad\qquad\qquad$ Apply transfer function

Let $\rho^{conc'} = \rho^{conc''} \Leftarrow \delta^{conc}$ $\qquad\qquad\qquad\qquad\qquad$ By rule $(\Leftarrow_{\rho})$

$\rho^{conc'} \sqsubseteq \rho^{abs'}$ $\qquad\qquad\qquad$ By Lemma 38($\Leftarrow_{\rho}$ preserves $\sqsubseteq$)

$f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}, instr) = \rho^{conc'}, \mathcal{A}^{conc''}$ $\qquad\qquad\qquad$ By rule (FLOW-CONS)

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad$ $\square$

Lemma 1 (Soundness of Single Constraint)**.**

$$\text{forall } deriv.$$
$$\mathcal{A}^{abs}; \rho^{abs}; cons \vdash instr \hookrightarrow \delta^{abs}, \gamma^{abs}$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$
$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$
$$\rho^{conc} \sqsubseteq \rho^{abs}$$
$$\mathcal{A}^{abs} \vdash \rho^{abs} \text{ consistent}$$
$$\mathcal{A}^{conc} \vdash \rho^{conc} \text{ consistent}$$

$$\text{exists } deriv.$$
$$\mathcal{A}^{conc}; \rho^{conc}; cons \vdash instr \hookrightarrow \delta^{conc}, \gamma^{conc}$$
$$\delta^{conc} \sqsubseteq \delta^{abs}$$
$$\gamma^{conc} \sqsubseteq \gamma^{abs}$$

Proof:
By case analysis on $\mathcal{A}^{abs}; \rho^{abs}; cons \vdash instr \hookrightarrow \delta^{abs}, \gamma^{abs}$

Case:
$$\frac{\begin{array}{c} instr : op \mapsto \beta \quad \Gamma_y = FV(op) \cup FV(P_{ctx}) \cup FV(\bar{Q}) \quad \mathsf{findLabels}(\mathcal{A}^{abs}; \Gamma_y; \beta) = \Sigma^{abs} \\ \Sigma^{abs} \neq \varnothing \quad \mathcal{T}^{abs} = \{\sigma, \delta, \gamma | \sigma \in \Sigma^{abs} \wedge \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \hookrightarrow \delta, \alpha \wedge \gamma = \alpha[\beta]\} \\ \mathcal{T}^{abs}.\sigma = \Sigma^{abs} \quad \delta^{abs} = \sqcup \mathcal{T}^{abs}.\delta \quad \gamma^{abs} = \sqcup \mathcal{T}^{abs}.\gamma \end{array}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; \vdash instr \hookrightarrow \delta^{abs}, \gamma^{abs}} \text{ (MATCH)}$$

$\Sigma^{conc} = \mathsf{findLabels}(\mathcal{A}^{conc}; \Gamma_y; \beta)$	By Lemma 22(FindLabels returns subsets)
$\Sigma^{conc} \subseteq \Sigma^{abs}$	By Lemma 22(FindLabels returns subsets)

By case analysis on Σ^{conc}

Case: $\Sigma^{conc} = \varnothing$

$\mathcal{A}^{conc}; \rho^{conc}; cons \vdash instr \hookrightarrow \mathsf{ignore}(\mathcal{A}^{conc}), \varnothing$	By rule (NO-MATCH)
$\varnothing \sqsubseteq \gamma^{conc}$	By rule $\sqsubseteq_\gamma -\varnothing$
$\mathcal{A}^{abs} \vdash \delta^{abs} \text{ consistent}$	By Lemma consistency of a single constraint
$\mathcal{A}^{conc} \vdash \bot(\mathcal{A}^{conc}) \text{ consistent}$	By Lemma $\bot$ is consistent
$\bot(\mathcal{A}^{conc}) \sqsubseteq \delta^{abs}$	By rule $\sqsubseteq_\delta -\delta$

Case: $\Sigma^{conc} \neq \varnothing$

$$\forall \sigma \in \Sigma^{conc}$$

$\mathcal{A}^{abs};\ \mathcal{B}^{abs};\ \rho^{abs};\ \sigma \vdash cons \hookrightarrow \delta^a, \alpha^a$ By $\sigma \in \Sigma^{abs}$

$\mathcal{A}^{conc} \vdash \sigma\ \mathsf{validFor}\ \Gamma_y$ By Lemma 22(FindLabels returns subsets)

$dom(\sigma) = dom(\Gamma_y)$ By Lemma 22(FindLabels returns subsets)

$\mathcal{A}^{conc};\ \mathcal{B}^{conc};\ \rho^{conc};\ \sigma \vdash cons \hookrightarrow \delta^c, \alpha^c$

          By Lemma 2(Soundness of Fully Bound Check)

$\delta^c \sqsubseteq \delta^a$ By Lemma 2(Soundness of Fully Bound Check)

$\alpha^c \sqsubseteq \alpha^a$ By Lemma 2(Soundness of Fully Bound Check)

$dom(\alpha^c) = dom(\sigma)$ By Lemma bound constraint check consistent

$\mathcal{A}^{conc} \vdash \delta^c\ \mathsf{consistent}$ By Lemma bound constraint check consistent

Let $\gamma^c = \alpha^c[\beta]$

$dom(\gamma^c) \subseteq dom(\mathcal{L}^{conc})$ By $dom(\beta) \subseteq dom(\mathcal{L}^{conc})$

$\gamma^c \sqsubseteq \gamma^a$ By Lemma subs preserves $\sqsubseteq$

Let $\mathcal{T}^{conc} = \{\sigma, \delta, \gamma \mid \sigma \in \Sigma^{conc} \wedge \mathcal{A}^{conc};\ \mathcal{B}^{conc};\ \rho^{conc};\ \sigma \vdash cons \hookrightarrow \delta, \alpha \wedge \gamma = \alpha[\beta]\}$

$\mathcal{T}^{conc}.\sigma = \Sigma^{conc}$ By set construction and quantifier

Let $\delta^{conc} = \sqcup\mathcal{T}^{conc}.\delta$

$\delta^{conc} \sqsubseteq \delta^{abs}$ By Lemma $\sqcup_\delta$ preserves $\sqsubseteq$ and Lemma $\sqcup_\delta$ is less precise than operands

Let $\gamma^{conc} = \sqcup\mathcal{T}^{conc}.\gamma$

$\gamma^{conc} \sqsubseteq \gamma^{abs}$ By Lemma $\sqcup_\delta$ preserves $\sqsubseteq$ and Lemma $\sqcup_\delta$ is less precise than operands

Case:
$$\dfrac{instr : op \mapsto \beta \qquad \mathsf{findLabels}(\mathcal{A}^{abs}; \Gamma_y; \beta) = \varnothing}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; \vdash instr \hookrightarrow \bot(\mathcal{A}^{abs}), \varnothing}\ \text{(NO–ALIASES)}$$

$\Sigma^{conc} = \mathsf{findLabels}(\mathcal{A}^{conc}; \Gamma_y; \beta)$ By Lemma 22(FindLabels returns subsets)

$\Sigma^{conc} \subseteq \Sigma^{abs}$ By Lemma 22(FindLabels returns subsets)

$\Sigma^{conc} = \varnothing$ By $\Sigma^{conc} \subset \Sigma^{abs}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; \vdash instr \hookrightarrow \bot(\mathcal{A}^{conc}), \varnothing$ By rule (NO-ALIASES)

$\bot(\mathcal{A}^{conc}) \sqsubseteq \bot(\mathcal{A}^{abs})$ By rule $\sqsubseteq -\bot$

$\varnothing \sqsubseteq \varnothing$ By rule $\sqsubseteq -\varnothing$

Case:
$$\dfrac{\neg(instr : op \mapsto \beta)}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; \vdash instr \hookrightarrow \bot(\mathcal{A}^{abs}), \varnothing}\ \text{(NO–MATCH)}$$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; \vdash instr \hookrightarrow \bot(\mathcal{A}^{conc}), \varnothing$ By rule (NO-MATCH)

$\bot(\mathcal{A}^{conc}) \sqsubseteq \bot(\mathcal{A}^{abs})$ By rule $\sqsubseteq -\bot$

$\varnothing \sqsubseteq \varnothing$ By rule $\sqsubseteq -\varnothing$

                        $\square$

Lemma 2 (Soundness of Fully Bound Check)**.**

$$\text{forall deriv.}$$

$$\text{cons} = \text{op} : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst}$$

$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$

$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$

$$\rho^{conc} \sqsubseteq \rho^{abs}$$

$$\mathcal{A}^{abs} \vdash \rho^{abs} \text{ consistent}$$

$$\mathcal{A}^{conc} \vdash \rho^{conc} \text{ consistent}$$

$$\mathcal{A}^{conc} \vdash \sigma \text{ validFor } FV(P_{ctx})$$

$$\text{dom}(\sigma) = \text{dom}(FV(P_{ctx}))$$

$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash \text{cons} \hookrightarrow \delta^{abs}, \alpha^{abs}$$

$$\text{exists deriv.}$$

$$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash \text{cons} \hookrightarrow \delta^{conc}, \alpha^{conc}$$

$$\delta^{conc} \sqsubseteq \delta^{abs}$$

$$\alpha^{conc} \sqsubseteq \alpha^{abs}$$

Proof:
By case analysis on $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash \text{cons} \hookrightarrow \delta^{abs}, \alpha^{abs}$

Case: $$\dfrac{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{ctx}[\sigma] \text{ False}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash \text{cons} \hookrightarrow \perp(\mathcal{A}^{abs}), \sigma} \text{(BOUND–F–SOUND)}$$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{ctx}[\sigma] \ t^c$	By Lemma 4(Truth Checking Sound)
$t^c \sqsubseteq \text{False}$	By Lemma 4(Truth Checking Sound)
$t^c = \text{False}$	By inversion on $t^c \sqsubseteq \text{False}$
$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash \text{cons} \hookrightarrow \perp(\mathcal{A}^{conc}), \varnothing$	By rule (BOUND-F-SOUND)
$\perp(\mathcal{A}^{conc}) \sqsubseteq \perp(\mathcal{A}^{abs})$	By Lemma $\perp$ preserves $\sqsubseteq$
$\sigma \sqsubseteq \sigma$	By rule ($\sqsubseteq -\varnothing$)

Case: $$\dfrac{\begin{array}{c} \text{cons} = \text{op} : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \\ \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{ctx}[\sigma] \text{ True} \qquad \text{allValidSubs}(\mathcal{A}^{abs}; \sigma; FV(\text{cons})) = \Sigma^{abs} \\ \forall \, \sigma' \in \Sigma^{abs} . \, \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho \vdash P_{req}[\sigma'] \text{ True} \\ \text{lattice}(\mathcal{A}^{abs}; \mathcal{B}^{abs}; \sigma; \bar{Q}) = \delta^{abs} \qquad \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_\alpha P_{rst} \hookrightarrow \alpha^{abs} \end{array}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash \text{cons} \hookrightarrow \delta^{abs}, \alpha^{abs}} \text{(BOUND–T–SOUND)}$$

$\mathcal{B}^{conc}; \rho^{conc} \vdash P_{ctx}[\sigma] \ t^c$	By Lemma 4(Truth Checking Sound)
$t^c \sqsubseteq \text{True}$	By Lemma 4(Truth Checking Sound)

$t^c = \mathsf{True}$ By inversion on $t^c \sqsubseteq \mathsf{True}$

$\Sigma^{conc} = \mathsf{allValidSubs}(\mathcal{A}^{conc}; \sigma; \mathsf{FV}(cons))$ By Lemma 23(ValidSubs returns subsets)

$\forall\, \sigma \in \Sigma^{conc}\,.\, \mathcal{A}^{conc} \vdash \sigma\ \mathsf{validFor}\ \mathsf{FV}(cons)$ By Lemma 23(ValidSubs returns subsets)

$\forall\, \sigma \in \Sigma^{conc}\,.\, \mathcal{A}^{conc} \vdash \sigma\ \mathsf{validFor}\ \mathsf{FV}(P_{req})$ By $\mathsf{FV}(P_{req}) \subseteq \mathsf{FV}(cons)$

$\Sigma^{conc} \subseteq \Sigma^{abs}$ By Lemma 23(ValidSubs returns subsets)

$\forall\, \sigma' \in \Sigma^{conc}\,.\, \mathcal{B}^{abs}; \rho^{abs} \vdash P_{req}[\sigma']\ \mathsf{True}$ By $\Sigma^c \subseteq \Sigma^a$

$\mathsf{lattice}(\mathcal{A}^{conc}; \mathcal{B}^{conc}; \sigma; \bar{Q}) = \delta^{conc}$ By Lemma 25(Lattice preserves precision)

$\delta^{conc} \sqsubseteq \delta^{abs}$ By Lemma 25(Lattice preserves precision)

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P_{rst} \hookrightarrow \alpha^{conc}$ By Lemma 3(Soundness of Restriction)

$\alpha^{conc} \sqsubseteq \alpha^{abs}$ By Lemma 3(Soundness of Restriction)

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_{full} cons \rightarrow \delta^{conc}, \sigma$ By rule (BOUND-T-SOUND)

Case:
$$\frac{\begin{array}{c} cons = op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \\ \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{ctx}[\sigma]\ \mathsf{Unknown} \\ \mathsf{allValidSubs}(\mathcal{A}^{abs}; \sigma; \mathsf{FV}(cons)) = \Sigma^{abs} \\ \forall\, \sigma' \in \Sigma^{abs}\,.\, \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho \vdash P_{req}[\sigma']\ \mathsf{True} \\ \mathsf{lattice}(\mathcal{A}^{abs}; \mathcal{B}^{abs}; \sigma; \bar{Q}) = \delta^{abs} \end{array}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \hookrightarrow^{\uparrow}_* \delta^{abs'}, \sigma}\ (\textsc{bound--u--sound})$$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{ctx}[\sigma]\ t^c$ By Lemma 4(Truth Checking Sound)

Case analysis on t^c

Case: $t^c = \mathsf{True}$

$\Sigma^{conc} = \mathsf{allValidSubs}(\mathcal{A}^{conc}; \sigma; \mathsf{FV}(cons))$ By Lemma 23(ValidSubs returns subsets)

$\forall\, \sigma \in \Sigma^{conc}\,.\, \mathcal{A}^{conc} \vdash \sigma\ \mathsf{validFor}\ \mathsf{FV}(cons)$ By Lemma 23(ValidSubs returns subsets)

$\Sigma^{conc} \subseteq \Sigma^{abs}$ By Lemma 23(ValidSubs returns subsets)

$\forall\, \sigma' \in \Sigma^{conc}\,.\, \mathcal{B}^{conc}; \rho^{conc} \vdash P_{req}[\sigma']\ \mathsf{True}$

By $\Sigma^c \subseteq \Sigma^a$ and Lemma 4(Truth Checking Sound)

$\mathsf{lattice}(\mathcal{A}^{conc}; \mathcal{B}^{conc}; \sigma; \bar{Q}) = \delta^{conc}$ By Lemma 25(Lattice preserves precision)

$\delta^{conc} \sqsubseteq \delta^{abs'}$ By Lemma 25(Lattice preserves precision)

Let $\delta^{abs} =^{\uparrow}_* \delta^{abs'}$

$\delta^{abs'} \sqsubseteq \delta^{abs}$ By Lemma $\overset{\uparrow}{*}$ result is less precise

$\delta^{conc} \sqsubseteq \delta^{abs}$ By Lemma transitivity of $\sqsubseteq$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P_{rst} \hookrightarrow \alpha^{conc}$By Lemma 24(Restriction less precise than substitution)

$\alpha^{conc} \sqsubseteq \sigma$ By Lemma 24(Restriction less precise than substitution)

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash cons \hookrightarrow \delta^{conc}, \alpha^{conc}$ By rule (BOUND-T-SOUND)

Case: $t^c = \mathsf{Unknown}$

$\Sigma^{conc} = \mathsf{allValidSubs}(\mathcal{A}^{conc}; \sigma; \mathsf{FV}(cons))$ By Lemma 23(ValidSubs returns subsets)

$\forall\, \sigma \in \Sigma^{conc}\,.\, \mathcal{A}^{conc} \vdash \sigma\ \mathsf{validFor}\ FV(cons)$ By Lemma 23(ValidSubs returns subsets)

$\Sigma^{conc} \subseteq \Sigma^{abs}$ By Lemma 23(ValidSubs returns subsets)

$\forall\, \sigma' \in \Sigma^{conc}\,.\, \mathcal{B}^{conc}; \rho^{conc} \vdash P_{req}[\sigma']\ \mathsf{True}$

         By $\Sigma^c \subseteq \Sigma^a$ and Lemma 4(Truth Checking Sound)

$\mathsf{lattice}(\mathcal{A}^{conc}; \mathcal{B}^{conc}; \sigma; \bar{Q}) = \delta^{conc\prime}$ By Lemma 25(Lattice preserves precision)

$\delta^{conc\prime} \sqsubseteq \delta^{abs\prime}$ By By Lemma 25(Lattice preserves precision)

Let $\delta^{abs} =_{*}^{\uparrow} \delta^{abs\prime}$

Let $\delta^{conc} =_{*}^{\uparrow} \delta^{conc\prime}$

$\delta^{conc} \sqsubseteq \delta^{abs}$ By Lemma $\overset{\uparrow}{*}$ preserves $\sqsubseteq$

$\sigma \sqsubseteq \sigma$ By rule $\sqsubseteq_{\alpha} - =$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash cons \hookrightarrow \delta^{conc}, \sigma$ By rule (BOUND-U-SOUND)

Case: $t^c = \mathsf{False}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash cons \hookrightarrow \bot(\mathcal{A}^{conc}), \sigma$ By rule (BOUND-F-SOUND)

$\bot(\mathcal{A}^{conc}) \sqsubseteq \delta^{abs}$ By rule $\sqsubseteq -\bot$

$\sigma \sqsubseteq \sigma$ By rule $\sqsubseteq - =$

$\square$

Lemma 3 (Soundness of Restriction)**.**

$forall\ deriv.$

$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_{\alpha} P \hookrightarrow \alpha^{abs}$$

$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$

$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$

$$\rho^{conc} \sqsubseteq \rho^{abs}$$

$$\mathcal{A}^{abs} \vdash \rho^{abs}\ \mathsf{consistent}$$

$$\mathcal{A}^{conc} \vdash \rho^{conc}\ \mathsf{consistent}$$

$exists\ deriv.$

$$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_{\alpha} P \hookrightarrow \alpha^{conc}$$

$$\alpha^{conc} \sqsubseteq \alpha^{abs}$$

Proof:

By case analysis on $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_{\alpha} P \hookrightarrow \alpha^{abs}$

Case:
$$\frac{\Sigma^{abs} = \mathsf{allValidSubs}(\mathcal{A}^{abs}; \sigma, FV(P)) \qquad \exists \sigma' \in \Sigma^{abs}.\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma']\ t^a \qquad t^a \neq \mathsf{False}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_{\alpha} P \hookrightarrow \sigma} (\text{RESTRICT}-\text{T}-\text{U}-\text{SOUND/COMPLETE})$$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P \hookrightarrow \alpha^{conc}$ By Lemma 24(Restriction less precise than substitution)

$\alpha^{conc} \sqsubseteq \sigma$ By Lemma 24(Restriction less precise than substitution)

Case: $\dfrac{\Sigma^{abs} = \mathsf{allValidSubs}(\mathcal{A}^{abs}; \sigma, FV(P)) \qquad \forall \sigma' \in \Sigma^{abs}.\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma'] \, \mathsf{False}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_\alpha P \hookrightarrow \bot(\sigma)}$ (RESTRICT−F−SOUND/COM

$\Sigma^{conc} = \mathsf{allValidSubs}(\mathcal{A}^{conc}; \sigma, FV(P))$ By Lemma 23(ValidSubs returns subsets)

$\Sigma^{conc} \subseteq \Sigma^{abs}$ By Lemma 23(ValidSubs returns subsets)

$\forall \sigma' \in \Sigma^{conc}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma'] \, \mathsf{False}$ By $\sigma' \in \Sigma^{abs}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P[\sigma'] \, \mathsf{False}$ By Lemma 4(Truth Checking Sound)

$\forall \sigma' \in \Sigma^{conc}.\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P[\sigma'] \, \mathsf{False}$ By quantification above

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P \hookrightarrow \bot(\sigma)$ By rule (RESTRICT-F-SOUND/COMPLETE)

$\bot(\sigma) \sqsubseteq \bot(\sigma)$ By $\sqsubseteq_\alpha - =$

$\square$

Lemma 4 (Truth Checking Sound)**.**

forall deriv.

$$\rho^{conc} \sqsubseteq \rho^{abs}$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$
$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$
$$\mathcal{A}^{conc} \vdash \sigma \, \mathsf{validFor} \, FV(P)$$
$$\mathcal{A}^{conc} \vdash \rho^{conc} \, \mathsf{consistent}$$
$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma] \, t^a$$

exists deriv.

$$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P[\sigma] \, t^c$$
$$t^c \sqsubseteq t^a$$

Proof:

By induction on $\rho^{abs} \vdash P[\sigma] \, t^a$

Case: $\dfrac{\rho^{abs}(\mathsf{rel}(\overline{y})[\sigma]) = t^a}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \mathsf{rel}(\overline{y})[\sigma] \, t^a}$ (REL)

Let $R = \mathsf{rel}(\bar{y})[\sigma]$

$R \in \mathsf{dom}(\rho^{conc})$ By Lemma σ valid and ρ consistent

Let $t^c = \rho^{conc}(R)$

$t^c \sqsubseteq t^a$ By inversion on $\rho^{conc} \sqsubseteq \rho^{abs}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \mathsf{rel}(\bar{y})[\sigma]\ t^c$ By rule (REL)

Case:
$$\frac{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho \vdash S[\sigma]\ t^a \qquad \mathcal{B}^{abs}(y_{test}[\sigma]) = t^a \qquad t^a \neq \mathsf{Unknown}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S/y_{test}[\sigma]\ \mathsf{True}}\,(\text{REL}-\text{TEST}-\text{T})$$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S[\sigma]\ t^c$ By induction hypothesis

$t^c \sqsubseteq t^a$

By case analysis on t^c

    Case: $t^c = \mathsf{True}$

        $\mathcal{B}^{conc}(y_{test}[\sigma]) = \mathsf{True}$ By $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

        $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma]\ \mathsf{True}$ By rule (REL $-$ TEST $-$ T)

        $\mathsf{True} \sqsubseteq \mathsf{True}$ By rule $\sqsubseteq\, -\, =$

    Case: $t^c = \mathsf{False}$

        $\mathcal{B}^{conc}(y_{test}[\sigma]) = \mathsf{False}$ By $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

        $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma]\ \mathsf{True}$ By rule (REL $-$ TEST $-$ T)

        $\mathsf{True} \sqsubseteq \mathsf{True}$ By rule $\sqsubseteq\, -\, =$

    Case: $t^c = \mathsf{Unknown}$

        Contradiction with $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

Case:
$$\frac{\begin{array}{cc} \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho \vdash S[\sigma]\ t_1^a \qquad \mathcal{B}^{abs}(y_{test}[\sigma]) = t_2^a \\ t_1^a \neq \mathsf{Unknown} \qquad t_2^a \neq \mathsf{Unknown} \qquad t_1^a \neq t_2^a \end{array}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S/y_{test}[\sigma]\ \mathsf{False}}\,(\text{REL}-\text{TEST}-\text{F})$$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S[\sigma]\ t_1^c$ By induction hypothesis

$t_1^c \sqsubseteq t_1^a$

By case analysis on t_1^c

    Case: $t_1^c = \mathsf{True}$

        $\mathcal{B}^{conc}(y_{test}[\sigma]) = t_2^c$ By $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

        $t_2^c = \mathsf{False}$ By $t_1^c \sqsubseteq t_1^a$ and $t_1^a \neq t_2^a$ and $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

        $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A/\ell_{test}\ \mathsf{False}$ By rule (REL $-$ TEST $-$ F)

        $\mathsf{False} \sqsubseteq \mathsf{False}$ By rule $\sqsubseteq\, -\, =$

Case: $t_1^c = \mathsf{False}$

$\qquad \mathcal{B}^{conc}(y_{test}[\sigma]) = t_2^c$ $\hfill$ By $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

$\qquad t_2^c = \mathsf{False}$ $\hfill$ By $t_1^c \sqsubseteq t_1^a$ and $t_1^a \neq t_2^a$ and $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

$\qquad \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A/\ell_{test} \; \mathsf{False}$ $\hfill$ By rule $(\mathrm{REL-TEST-F})$

$\qquad \mathsf{False} \sqsubseteq \mathsf{False}$ $\hfill$ By rule $\sqsubseteq - \, =$

Case: $t_1^c = \mathsf{Unknown}$

$\qquad$ Contradiction with $\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$

Case: $\dfrac{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S[\sigma] \; \mathsf{Unknown}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S/y_{test}[\sigma] \; \mathsf{Unknown}} (\mathrm{REL-TEST-U1})$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S[\sigma] \; t^c$ $\hfill$ By induction hypothesis

$t_1^c \sqsubseteq \mathsf{Unknown}$ $\hfill$ By induction hypothesis

Let $t_2^c = \mathcal{B}^{conc}(\ell_{test})$ By case analysis on t_1^c

Case: $t_1^c = \mathsf{True}$

$\qquad$ Let $t_2^c = \mathcal{B}^{conc}(y_{test}[\sigma])$

$\qquad$ By case analysis on t_1^c

$\qquad$ **Case:** $t_2^c = \mathsf{True}$

$\qquad\qquad \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \; \mathsf{True}$ $\hfill$ By rule $(\mathrm{REL-TEST-T})$

$\qquad\qquad \mathsf{True} \sqsubseteq \mathsf{Unknown}$ $\hfill$ By rule $\sqsubseteq -\top$

$\qquad$ **Case:** $t_2^c = \mathsf{False}$

$\qquad\qquad \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \; \mathsf{False}$ $\hfill$ By rule $(\mathrm{REL-TEST-F})$

$\qquad\qquad \mathsf{False} \sqsubseteq \mathsf{Unknown}$ $\hfill$ By rule $\sqsubseteq -\top$

$\qquad$ **Case:** $t_2^c = \mathsf{Unknown}$

$\qquad\qquad \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \; \mathsf{Unknown}$ $\hfill$ By rule $(\mathrm{REL-TEST-U2})$

$\qquad\qquad \mathsf{Unknown} \sqsubseteq \mathsf{Unknown}$ $\hfill$ By rule $\sqsubseteq -\top$

Case: $t_1^c = \mathsf{False}$

$\qquad$ Let $t_2^c = \mathcal{B}^{conc}(y_{test}[\sigma])$

$\qquad$ By case analysis on t_1^c

Case: $t_2^c = \text{False}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \text{ True}$	By rule (REL − TEST − T)
$\text{True} \sqsubseteq \text{Unknown}$	By rule $\sqsubseteq -\top$

Case: $t_2^c = \text{True}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \text{ False}$	By rule (REL − TEST − F)
$\text{False} \sqsubseteq \text{Unknown}$	By rule $\sqsubseteq -\top$

Case: $t_2^c = \text{Unknown}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \text{ Unknown}$	By rule (REL − TEST − U2)
$\text{Unknown} \sqsubseteq \text{Unknown}$	By rule $\sqsubseteq -\top$

Case: $t_1^c = \text{Unknown}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \text{ Unknown}$	By rule (REL − TEST − U1)
$\text{Unknown} \sqsubseteq \text{Unknown}$	By rule $\sqsubseteq -\top$

Case:
$$\frac{\mathcal{B}^{abs}(y_{test}[\sigma]) = \text{Unknown} \qquad \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S[\sigma] \ t^a}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S/y_{test}[\sigma] \text{ Unknown}} (\text{REL}-\text{TEST}-\text{U2})$$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A \ t_1^c$	By induction hypothesis
$t_1^c \sqsubseteq t^a$	By induction hypothesis
By case analysis on t_1^c	

Case: $t_1^c = \text{True}$

Let $t_2^c = \mathcal{B}^{conc}(y_{test}[\sigma])$
By case analysis on t_2^c

Case: $t_2^c = \text{True}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \text{ True}$	By rule (REL − TEST − T)
$\text{True} \sqsubseteq \text{Unknown}$	By rule $\sqsubseteq -\top$

Case: $t_2^c = \text{False}$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \text{ False}$	By rule (REL − TEST − F)
$\text{False} \sqsubseteq \text{Unknown}$	By rule $\sqsubseteq -\top$

Case: $t_2^c =$ Unknown

$\qquad$ $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma]$ Unknown $\qquad\qquad$ By rule (REL − TEST − U2)
$\qquad$ Unknown $\sqsubseteq$ Unknown $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By rule $\sqsubseteq -\top$

Case: $t_1^c =$ False

$\qquad$ Let $t_2^c = \mathcal{B}^{conc}(y_{test}[\sigma])$
$\qquad$ By case analysis on t_1^c

$\qquad$ **Case:** $t_2^c =$ False

$\qquad\qquad$ $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma]$ True $\qquad\qquad$ By rule (REL − TEST − T)
$\qquad\qquad$ True $\sqsubseteq$ Unknown $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By rule $\sqsubseteq -\top$

$\qquad$ **Case:** $t_2^c =$ True

$\qquad\qquad$ $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma]$ False $\qquad\qquad$ By rule (REL − TEST − F)
$\qquad\qquad$ False $\sqsubseteq$ Unknown $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By rule $\sqsubseteq -\top$

$\qquad$ **Case:** $t_2^c =$ Unknown

$\qquad\qquad$ $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma]$ Unknown $\qquad$ By rule (REL − TEST − U2)
$\qquad\qquad$ Unknown $\sqsubseteq$ Unknown $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By rule $\sqsubseteq -\top$

Case: $t_1^c =$ Unknown

$\qquad$ $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma]$ Unknown $\qquad\qquad$ By rule (REL − TEST − U1)
$\qquad$ Unknown $\sqsubseteq$ Unknown $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By rule $\sqsubseteq -\top$

Case: $\dfrac{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A[\sigma] \text{ Unknown}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg A[\sigma] \text{ Unknown}}$ (¬T−UNKNOWN)

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A[\sigma]\ t^c$ $\qquad\qquad\qquad\qquad\qquad\qquad$ By induction hypothesis
$t^c \sqsubseteq$ Unknown $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By induction hypothesis
By case analysis on the value of t^c

$\qquad$ **Case:** $t^c =$ True

$\qquad\qquad$ $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A$ False $\qquad\qquad\qquad\qquad$ By rule (¬ − T − F)
$\qquad\qquad$ False $\sqsubseteq$ Unknown $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By rule $\sqsubseteq -\top$

Case: $t^c = $ False

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A$ True   By rule $(\neg - \top - \top)$
True $\sqsubseteq$ Unknown   By rule $\sqsubseteq - \top$

Case: $t^c = $ Unknown

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A$ Unknown   By rule $(\neg - \top - \cup)$
Unknown $\sqsubseteq$ Unknown   By rule $\sqsubseteq - \top$

Case: $\dfrac{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A[\sigma]\text{False}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg A[\sigma]\text{True}}(\neg \text{-} \top \text{-} \top)$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A[\sigma]\ t^c$ By induction hypothesis
$t^c \sqsubseteq$ False   By induction hypothesis
$t^c = $ False   By inversion on $t^c \sqsubseteq$ False
$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A[\sigma]$ True   By rule $(\neg - \top - \top)$

Case: $\dfrac{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A[\sigma]\text{True}}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg A[\sigma]\text{False}}(\neg \text{-} \top \text{-} \text{F})$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A[\sigma]\ t^c$ By induction hypothesis
$t^c \sqsubseteq$ True   By induction hypothesis
$t^c = $ True   By inversion on $t^c \sqsubseteq$ True
$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A[\sigma]$ False   By rule $(\neg - \top - \text{F})$

Case: $\dfrac{}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \textit{true}\ \text{True}}(\text{TRUE})$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \textit{true}\ \text{True}$ By rule (TRUE)
True $\sqsubseteq$ True   By rule $\sqsubseteq - =$

Case: $\dfrac{}{\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \textit{false}\ \text{False}}(\text{FALSE})$

$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \textit{false}\ \text{False}$ By rule (FALSE)
False $\sqsubseteq$ False   By rule $\sqsubseteq - =$

Remaining cases work as expected for a three value logic.

$\square$

C.2 Completeness

Theorem 2. Completeness of Relations Analysis

$$\text{forall der.}$$

$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$

$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$

$$\mathcal{A}^{abs} \vdash \rho^{abs} \text{ consistent}$$

$$\mathcal{A}^{conc} \vdash \rho^{conc} \text{ consistent}$$

$$\rho^{conc} \sqsubseteq \rho^{abs}$$

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

$$\text{exists der.}$$

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

$$\rho^{conc'} \sqsubseteq \rho^{abs'}$$

$$\mathcal{A}^{conc''} \sqsubseteq \mathcal{A}^{abs''}$$

Proof:

$\mathcal{T}^{conc} = \{cons, \delta, \gamma | cons \in \mathcal{C} \wedge \mathcal{A}^{conc'}; \mathcal{B}^{conc}; \rho^{conc}; cons \vdash instr \hookrightarrow \delta, \gamma\}$

	By inversion on $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; instr) = \rho^{conc'}, \mathcal{A}^{conc''}$
$\mathcal{T}^{conc}.cons = \mathcal{C}$	By inversion on $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; instr) = \rho^{conc'}, \mathcal{A}^{conc''}$
$\gamma^{conc} = \sqcup \mathcal{T}^{conc}.\gamma$	By inversion on $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; instr) = \rho^{conc'}, \mathcal{A}^{conc''}$
$\mathcal{A}^{conc''} = \mathcal{A}^{conc'} \Leftarrow \gamma^{conc}$	By inversion on $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; instr) = \rho^{conc'}, \mathcal{A}^{conc''}$
$\delta^{conc} = \boxminus \mathcal{T}^{conc}.\delta$	By inversion on $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; instr) = \rho^{conc'}, \mathcal{A}^{conc''}$
$\rho^{conc'} = \mathsf{transfer}(\rho^{conc}, \mathcal{A}^{conc''}) \Leftarrow \delta^{conc}$	
	By inversion on $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; instr) = \rho^{conc'}, \mathcal{A}^{conc''}$
$f_{alias}(\mathcal{A}^{conc}; instr) = \mathcal{A}^{conc'}$	By inversion on $f_{\mathcal{C};\mathcal{A}^{conc};\mathcal{B}^{conc}}(\rho^{conc}; instr) = \rho^{conc'}, \mathcal{A}^{conc''}$
$f_{alias}(\mathcal{A}^{abs}; instr) = \mathcal{A}^{abs'}$	By Theorem Aliasing Flow Function Complete
$\mathcal{A}^{conc'} \sqsubseteq_A \mathcal{A}^{abs'}$	By Theorem Aliasing Flow Function Complete
$\mathcal{A}^{abs'} \vdash \rho^{abs'}$ consistent	By Theorem Aliasing Flow Function Preserves Consistency

$\forall\, cons \in \mathcal{C} \,.$

Let $\mathcal{A}^{conc'}; \mathcal{B}^{conc}; \rho^{conc}; cons \vdash instr \hookrightarrow \delta^c, \gamma^c$	By construction of $\mathcal{T}^{conc}$
$\mathcal{A}^{abs'}; \mathcal{B}^{abs}; \rho^{abs}; cons \vdash instr \hookrightarrow \delta^a, \gamma^a$	By Lemma 5(Completeness of Single Constraint)
$\delta^c \sqsubseteq \delta^a$	By Lemma 5(Completeness of Single Constraint)
$\gamma^c \sqsubseteq \gamma^a$	By Lemma 5(Completeness of Single Constraint)
$\mathcal{A}^{abs'} \vdash \delta^a$ consistent	By Lemma 9(Consistency of a Single Constraint)
$dom(\gamma^a) \subseteq dom(\mathcal{A}^{abs'}.\mathcal{L})$	By Lemma 9(Consistency of a Single Constraint)

Let $\mathcal{T}^{abs} = \{cons, \delta, \gamma \mid cons \in \mathcal{C} \wedge \mathcal{A}^{abs'}; \mathcal{B}^{abs}; \rho^{abs}; cons \vdash instr \hookrightarrow \delta, \gamma\}$

$\mathcal{T}^{conc}.cons = \mathcal{C}$	By set construction
Let $\delta^{abs} = \sqcup \mathcal{T}^{abs}.\delta$	By rule (EQJOIN)

$\delta^{conc} \sqsubseteq \delta^{abs}$	By Lemma 30(eqjoin delta operator preserves alap)
Let $\gamma^{abs} = \sqcup \mathcal{T}^{abs}.\gamma$	By rule $(\sqcup_\gamma)$
$\gamma^{conc} \sqsubseteq \gamma^{abs}$	By Lemma 33($\sqcup_\gamma$ operator preserves $\sqsubseteq$)
Let $\mathcal{A}^{abs''} = \mathcal{A}^{abs'} \Leftarrow \gamma$	By rule $(\Leftarrow_\mathcal{A})$
$\mathcal{A}^{conc''} \sqsubseteq \mathcal{A}^{abs''}$	By Lemma 34(Leftarrow AliasA preserves alap)
Let $\rho^{abs''} = \textsf{transfer}(\rho^{abs}, \mathcal{A}^{abs''})$	Apply function $\textsf{transfer}$
Let $\rho^{abs'} = \rho^{abs''} \Leftarrow \delta^{abs}$	By rule $(\Leftarrow_\rho)$
$\rho^{conc'} \sqsubseteq \rho^{abs'}$	By Lemma 38($\Leftarrow_\rho$ preserves $\sqsubseteq$)
$f_{\mathbb{C}}(\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}, instr) = \rho^{abs'}, \mathcal{A}^{abs''}$	By rule (FLOW-CONS)

$\square$

Lemma 5 (Completeness of Single Constraint)**.**

$$forall\ deriv.$$

$$\mathcal{A}^{conc}; \rho^{conc}; cons \vdash instr \hookrightarrow \delta^{conc}, \gamma^{conc}$$

$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$

$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$

$$\rho^{conc} \sqsubseteq \rho^{abs}$$

$$\mathcal{A}^{abs} \vdash \rho^{abs}\ \textsf{consistent}$$

$$\mathcal{A}^{conc} \vdash \rho^{conc}\ \textsf{consistent}$$

$$exists\ deriv.$$

$$\mathcal{A}^{abs}; \rho^{abs}; cons \vdash instr \hookrightarrow \delta^{abs}, \gamma^{abs}$$

$$\delta^{conc} \sqsubseteq \delta^{abs}$$

$$\gamma^{conc} \sqsubseteq \gamma^{abs}$$

Proof:
By case analysis on $\mathcal{A}^{conc}; \rho^{conc}; cons \vdash instr \hookrightarrow \delta^{conc}, \gamma^{conc}$

Case:
$$\frac{\begin{array}{c} instr : op \mapsto \beta \\ \Gamma_y = FV(op) \cup FV(P_{ctx}) \cup FV(\bar{Q}) \qquad \textsf{findLabels}(\mathcal{A}^{conc}; \Gamma_y; \beta) = \Sigma^{conc} \qquad \Sigma^{conc} \neq \varnothing \\ \mathcal{T}^{conc} = \{\sigma, \delta, \gamma \mid \sigma \in \Sigma^{conc} \wedge \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash cons \hookrightarrow \delta, \alpha \wedge \gamma = \alpha[\beta]\} \\ \mathcal{T}^{conc}.\sigma = \Sigma^{conc} \qquad \delta^{conc} = \sqcup \mathcal{T}^{conc}.\delta \qquad \gamma^{conc} = \sqcup \mathcal{T}^{conc}.\gamma \end{array}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \delta^{conc}, \gamma^{conc}} \text{(MATCH)}$$

$\Sigma^{abs} = \textsf{findLabels}(\mathcal{A}^{abs}; \Gamma_y; \beta) = \Sigma^{abs}$	By Lemma 22(FindLabels returns subsets)
$\Sigma^{conc} \subseteq \Sigma^{abs}$	By Lemma 22(FindLabels returns subsets)
$\forall \sigma \in \Sigma^{abs}.\mathcal{A} \vdash \sigma\ \textsf{validFor}\ \Gamma_y \wedge dom(\sigma) = dom(\Gamma_y)$	By Lemma 22(FindLabels returns subsets)
$\Sigma^{abs} \neq \varnothing$	By $\Sigma^{conc} \neq \varnothing \wedge \Sigma^{conc} \subseteq \Sigma^{abs}$
$\forall \sigma \in \Sigma^{conc}$.	

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \hookrightarrow \delta^a, \alpha^a$ By Lemma 6(Completeness with Full Substitution)

$\delta^c \sqsubseteq \delta^a$ By Lemma 6(Completeness with Full Substitution)

$\alpha^c \sqsubseteq \alpha^a$ By Lemma 6(Completeness with Full Substitution)

$\alpha^c[\beta] \sqsubseteq \alpha^a[\beta]$ By Lemma substitution preserves $\sqsubseteq$

Let $\mathcal{T} = \{\sigma, \delta, \gamma \mid \sigma \in \Sigma^{abs} \land \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}\sigma \vdash \delta, \gamma \land \gamma = \alpha[\beta]\}$

$\mathcal{T}.\sigma = \Sigma^{abs}$ By Lemma bound passes when σ valid

Let $\delta^{abs} = \sqcup \mathcal{T}.\delta$

$\delta^{conc} \sqsubseteq \delta^{abs}$ By Lemma $\sqcup$ preserves $\sqsubseteq$ and Lemma $\sqcup$ less precise than operands

Let $\gamma^{abs} = \sqcup \mathcal{T}.\gamma$

$\gamma^{conc} \sqsubseteq \gamma^{abs}$ By Lemma $\sqcup$ preserves $\sqsubseteq$ and Lemma $\sqcup$ less precise than operands

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \delta^{abs}, \gamma^{abs}$ By rule (MATCH)

Case:
$$\frac{instr : op \mapsto \beta \qquad \mathsf{findLabels}(\mathcal{A}^{conc}; \Gamma_y; \beta) = \varnothing}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \perp(\mathcal{A}^{conc}), \varnothing} \text{(NO–ALIASES)}$$

Let $\Sigma^{abs} = \mathsf{findLabels}(\mathcal{A}^{abs}; \Gamma_y; \beta)$ By set construction

Case analysis on the structure of Σ^{abs}

Case: $\Sigma^{abs} = \varnothing$

    $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \perp(\mathcal{A}^{conc}), \varnothing$

                                               By rule (NO-ALIASES)

    $\perp(\mathcal{A}^{conc}) \sqsubseteq \perp(\mathcal{A}^{abs})$ By Lemma $\perp$ maintains $\sqsubseteq$

    $\varnothing \sqsubseteq \varnothing$ By rule $\sqsubseteq - \varnothing$

Case: $\Sigma^{abs} \neq \varnothing$

    Let $\mathcal{T} = \{\sigma, \delta, \gamma \mid \sigma \in \Sigma^{abs} \land \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}\sigma \vdash \delta, \gamma \land \gamma = \alpha[\beta]\}$

    $\mathcal{T}.\sigma = \Sigma^{abs}$ By Lemma bound passes when σ valid

    Let $\delta^{abs} = \sqcup \mathcal{T}.\delta$

    $\perp(\mathcal{A}^{conc}) \sqsubseteq \delta^{abs}$ By rule $\sqsubseteq - \perp$

    Let $\gamma^{abs} = \sqcup \mathcal{T}.\gamma$

    $\varnothing \sqsubseteq \gamma^{abs}$ By rule $\sqsubseteq - \varnothing$

    $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \delta^{abs}, \gamma^{abs}$ By rule (MATCH)

Case:
$$\frac{\neg(instr : op \mapsto \beta)}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \perp(\mathcal{A}^{conc}), \varnothing} \text{(NO–MATCH)}$$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \bot(\mathcal{A}^{abs}), \varnothing$ By rule (NO-MATCH)

$\bot(\mathcal{A}^{conc}) \sqsubseteq \bot(\mathcal{A}^{abs})$ By Lemma $\bot$ preserves $\sqsubseteq$

$\varnothing \sqsubseteq \varnothing$ By rule $\sqsubseteq -\varnothing$

$\square$

Lemma 6 (Completeness with Full Substitution)**.**

$$\text{forall deriv.}$$

$$\mathcal{A}^{conc} \sqsubseteq_{\mathcal{A}} \mathcal{A}^{abs}$$

$$\mathcal{B}^{conc} \sqsubseteq_{\mathcal{B}} \mathcal{B}^{abs}$$

$$\rho^{conc} \sqsubseteq \rho^{abs}$$

$$\mathcal{A}^{abs} \vdash \rho^{abs} \text{ consistent}$$

$$\mathcal{A}^{conc} \vdash \rho^{conc} \text{ consistent}$$

$$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash cons \rightarrow \delta^{conc}, \alpha^{conc}$$

$$\mathcal{A}^{conc} \vdash \sigma \text{ validFor } \Gamma_y$$

$$\Gamma_y = \Gamma_y = FV(op) \cup FV(P^{ctx}) \cup FV(\overline{Q})$$

$$\text{exists deriv.}$$

$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \rightarrow \delta^{abs}, \alpha^{abs}$$

$$\delta^{conc} \sqsubseteq \delta^{abs}$$

$$\alpha^{conc} \sqsubseteq \alpha^{abs}$$

Proof:
By case analysis on $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash cons \rightarrow \delta^{conc}, \alpha^{conc}$

Case:
$$\frac{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{ctx}[\sigma] \text{ False}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \hookrightarrow \bot(\mathcal{A}^{conc}), \sigma} (\text{BOUND}-\text{F})$$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{ctx}[\sigma] \text{ } t^a$	By Lemma 8(Truth Checking Complete)
$\text{False} \sqsubseteq t^a$	By Lemma 8(Truth Checking Complete)

By case analysis on the value of t^a

Case: $t^a = \text{False}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \hookrightarrow \bot(\mathcal{A}^{abs}), \sigma$	By rule (BOUND-F)
$\bot(\mathcal{A}^{conc}) \sqsubseteq \bot(\mathcal{A}^{abs})$	By Lemma $\bot$ preserves $\sqsubseteq$
$\sigma \sqsubseteq \sigma$	By rule $\sqsubseteq - =$

Case: $t^a = \text{True}$

Invalid case by $\text{False} \sqsubseteq t^a$

Case: $t^a = \text{Unknown}$

$\text{lattice}(\mathcal{A}^{abs}; \mathcal{B}^{abs}; \sigma; \bar{Q}) = \delta^{abs}$	By applying function lattice

Let $\delta^{abs} =^{\uparrow}_* \delta^{abs\prime}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \hookrightarrow \delta^{abs}, \sigma$ By rule (BOUND-U)

$\mathcal{A}^{abs} \vdash \delta^{abs\prime}$ consistent   By lattice consistent

$\mathcal{A}^{abs} \vdash \delta^{abs}$ consistent   By $\overset{\uparrow}{*}$ preserves consistent

$\mathcal{A}^{conc} \vdash \bot(\mathcal{A}^{conc})$ consistent   By Lemma $\bot$ is consistent

$\bot(\mathcal{A}^{conc}) \sqsubseteq \delta^{abs}$ By Lemma $\bot$ is less precise

$\sigma \sqsubseteq \sigma$ By rule $\sqsubseteq - =$

Case:

$$cons = op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst}$$
$$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{ctx}[\sigma] \; \text{True} \qquad allValidSubs(\mathcal{A}^{conc}; \sigma; FV(cons)) = \Sigma^{conc}$$
$$\exists \, \sigma' \in \Sigma^{conc} . \, \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{req}[\sigma'] \; \text{True} \vee \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{req}[\sigma'] \; \text{Unknown}$$
$$\frac{lattice(\mathcal{A}^{conc}; \mathcal{B}^{conc}; \sigma; \bar{Q}) = \delta^{conc} \qquad \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P_{rst} \hookrightarrow \alpha^{conc}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash cons \hookrightarrow \delta^{conc}, \alpha^{conc}} \text{(BOUND}-\text{T)}$$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{ctx}[\sigma] \; t^a$ By Lemma 8(Truth Checking Complete)

$\text{True} \sqsubseteq t^a$ By Lemma 8(Truth Checking Complete)

By case analysis on t^a

Case: $t^a = \text{True}$

$\Sigma^{abs} = allValidSubs(\mathcal{A}^{abs}; \sigma; FV(cons))$ By Lemma 23(ValidSubs returns subsets)

$\Sigma^{conc} \subseteq \Sigma^{abs}$ By Lemma 23(ValidSubs returns subsets)

$\exists \, \sigma' \in \Sigma^{abs} . \, \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{req}[\sigma'] \; \text{True} \vee$
$\quad \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{req}[\sigma'] \; \text{Unknown}$ By $\Sigma^{conc} \subseteq \Sigma^{abs}$

$\exists \, \sigma' \in \Sigma^{abs} . \, \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{req}[\sigma'] \; \text{True} \vee$
$\quad \mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{req}[\sigma'] \; \text{Unknown}$ By Lemma 8(Truth Checking Complete)

$lattice(\mathcal{A}^{abs}; \mathcal{B}^{abs}; \sigma; \bar{Q}) = \delta^{abs}$ By applying function lattice

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_\alpha P_{rst} \hookrightarrow \alpha^{abs}$ By Lemma 7(Completeness of Restriction)

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \rightarrow \delta^{abs}, \sigma$ By rule (BOUND-T)

$\delta^{conc} \sqsubseteq \delta^{abs}$ By Lemma 25(Lattice preserves precision)

$\alpha^{conc} \sqsubseteq \alpha^{abs}$ By Lemma 7(Completeness of Restriction)

Case: $t^a = \text{False}$

Invalid case by $\text{True} \sqsubseteq t^a$

Case: $t^a = \text{Unknown}$

$lattice(\mathcal{A}^{abs}; \mathcal{B}^{abs}; \sigma; \bar{Q}) = \delta^{abs}$ By applying function lattice

$\delta^{conc} \sqsubseteq \delta^{abs\prime}$ By Lemma 25(Lattice preserves precision)

Let $\delta^{abs} =^{\uparrow}_* \delta^{abs\prime}$

$$\delta^{abs'} \sqsubseteq \delta^{abs} \qquad\qquad\qquad \text{By Lemma polar less precise than operand}$$
$$\delta^{conc} \sqsubseteq \delta^{abs} \qquad\qquad\qquad \text{By Lemma transitivity of } \sqsubseteq$$
$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \hookrightarrow \delta^{abs}, \sigma \qquad\qquad \text{By rule (BOUND-U)}$$
$$\alpha^{conc} \sqsubseteq \sigma \qquad\qquad \text{By Lemma restrict less precise than substitution}$$

Case:
$$\frac{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P_{ctx}[\sigma] \text{ Unknown} \qquad \text{lattice}(\mathcal{A}^{conc}; \mathcal{B}^{conc}; \sigma; \bar{Q}) = \delta^{conc}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \overset{\uparrow}{\hookrightarrow}_* \delta^{conc}, \sigma} \text{(BOUND–U)}$$

$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{ctx}[\sigma]\, t^a \qquad\qquad \text{By Lemma 8(Truth Checking Complete)}$$
$$\text{Unknown} \sqsubseteq t^a \qquad\qquad \text{By Lemma 8(Truth Checking Complete)}$$
$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P_{ctx}[\sigma] \text{ Unknown} \qquad\qquad \text{By inversion on Unknown} \sqsubseteq t^a$$
$$\text{lattice}(\mathcal{A}^{abs}; \mathcal{B}^{abs}; \sigma; \bar{Q}) = \delta^{abs} \qquad\qquad \text{By applying function lattice}$$
$$\delta^{conc'} \sqsubseteq \delta^{abs'} \qquad\qquad \text{By Lemma 25(Lattice preserves precision)}$$
$$\overset{\uparrow}{*} \delta^{conc} \sqsubseteq \overset{\uparrow}{*} \delta^{abs} \qquad\qquad \text{By Lemma } \overset{\uparrow}{*} \text{ preserves } \sqsubseteq$$
$$\sigma \sqsubseteq \sigma \qquad\qquad \text{By rule } \sqsubseteq -=$$
$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash cons \hookrightarrow \delta^{abs}, \alpha^{abs} \qquad\qquad \text{By rule (BOUND-U)}$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square$$

Lemma 7 (Completeness of Restriction).

$$\texttt{forall } deriv.$$
$$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P \hookrightarrow \alpha^{conc}$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$
$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$
$$\rho^{conc} \sqsubseteq \rho^{abs}$$
$$\mathcal{A}^{abs} \vdash \rho^{abs} \text{ consistent}$$
$$\mathcal{A}^{conc} \vdash \rho^{conc} \text{ consistent}$$
$$\texttt{exists } deriv.$$
$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_\alpha P \hookrightarrow \alpha^{abs}$$
$$\alpha^{conc} \sqsubseteq \alpha^{abs}$$

Proof:
By case analysis on $\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P \hookrightarrow \alpha^{conc}$

Case:
$$\frac{\Sigma^{conc} = \text{allValidSubs}(\mathcal{A}^{conc}; \sigma, FV(P)) \qquad\quad}{\exists \sigma' \in \Sigma^{conc}.\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P[\sigma']\, t^c \qquad t^c \neq \text{False}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_\alpha P \hookrightarrow \sigma} \text{(RESTRICT–T–U–SOUND/COMPLETE)}$$

$\Sigma^{abs} = \mathsf{allValidSubs}(\mathcal{A}^{abs}; \sigma, \mathsf{FV}(P))$ — By applying function $\mathsf{allValidSubs}$

$\Sigma^{conc} \subseteq \Sigma^{conc}$ — By Lemma 23(ValidSubs returns subsets)

$\exists \sigma' \in \Sigma^{abs}.\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P[\sigma']\ t^c$ — By $\Sigma^{conc} \subseteq \Sigma^{conc}$

$\exists \sigma' \in \Sigma^{abs}.\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma']\ t^a$ — By Lemma 8(Truth Checking Complete)

$t^c \sqsubseteq t^a$ — By Lemma 8(Truth Checking Complete)

$t^c \neq \mathsf{False}$ — By $t^c \sqsubseteq t^a$ and $t^c \neq \mathsf{False}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_{\alpha} P \hookrightarrow \sigma$ — By rule (RESTRICT-T-U-SOUND/COMPLETE)

$\sigma \sqsubseteq \sigma$ — By rule ($\sqsubseteq - =$)

Case:
$$\dfrac{\Sigma^{conc} = \mathsf{allValidSubs}(\mathcal{A}^{conc}; \sigma, \mathsf{FV}(P)) \quad \forall \sigma' \in \Sigma^{conc}.\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash P[\sigma']\ \mathsf{False}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc}; \sigma \vdash_{\alpha} P \hookrightarrow \perp(\sigma)} \text{(RESTRICT}-\text{F}-\text{SOUND/COMPLETE)}$$

$\Sigma^{abs} = \mathsf{allValidSubs}(\mathcal{A}^{abs}; \sigma, \mathsf{FV}(P))$ — By applying function $\mathsf{allValidSubs}$

$\Sigma^{conc} \subseteq \Sigma^{abs}$ — By Lemma 23(ValidSubs returns subsets)

$\forall \sigma' \in \Sigma^{abs}.\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma']\ t^a$ — By Lemma consistency of truth checking

Case on property of $\Sigma^{a}bs$

Case: $\forall \sigma' \in \Sigma^{abs}.\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma']\ \mathsf{False}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_{\alpha} P \hookrightarrow \sigma$ — By rule (RESTRICT-F-SOUND/COMPLETE)

$\perp(\sigma) \sqsubseteq \sigma$ — By rule ($\sqsubseteq - \perp$)

Case: $\exists \sigma' \in \Sigma^{abs}.\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash P[\sigma']\ t^a\ \wedge t^a \neq \mathsf{False}$

$t^c \sqsubseteq t^a$ — By Lemma 8(Truth Checking Complete)

$t^c \neq \mathsf{False}$ — By $t^c \sqsubseteq t^a$ and $t^c \neq \mathsf{False}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs}; \sigma \vdash_{\alpha} P \hookrightarrow \sigma$ — By rule (RESTRICT-T-U-SOUND/COMPLETE)

$\perp(\sigma) \sqsubseteq \sigma$ — By rule ($\sqsubseteq - \perp$)

$\square$

Lemma 8 (Truth Checking Complete)**.**

$$forall\ deriv.$$

$$\rho^{conc} \sqsubseteq \rho^{abs}$$

$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$

$$\mathcal{B}^{conc} \sqsubseteq \mathcal{B}^{abs}$$

$$\mathcal{A}^{abs} \vdash \sigma\ \mathsf{validFor}\ \mathsf{FV}(\mathsf{P})$$

$$\mathcal{A}^{abs} \vdash \rho^{abs}\ \mathsf{consistent}$$

$$\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \mathsf{P}[\sigma]\mathsf{t}^{c}$$

$$exists\ deriv.$$

$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \mathsf{P}[\sigma]\mathsf{t}^{a}$$

$$\mathsf{t}^{c} \sqsubseteq \mathsf{t}^{a}$$

Proof:
By induction on $\rho^{conc} \vdash \mathsf{P}[\sigma]\ \mathsf{t}_{a}$

Case: $\dfrac{\rho^{conc}(\mathsf{rel}(\bar{\mathsf{y}})[\sigma]) = \mathsf{t}^{c}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \mathsf{rel}(\bar{\mathsf{y}})[\sigma]\ \mathsf{t}^{c}}\ \text{(REL)}$

$\quad$ Let $\mathsf{R} = \mathsf{rel}(\bar{\mathsf{y}})[\sigma]$
$\quad$ $\mathsf{R} \in \mathsf{dom}(\rho^{abs})$ $\qquad\qquad\qquad\qquad\qquad$ By Lemma σ valid and ρ consistent
$\quad$ Let $\mathsf{t}^{a} = \rho^{abs}(\mathsf{R})$
$\quad$ $\mathsf{t}^{c} \sqsubseteq \mathsf{t}^{a}$ $\qquad\qquad\qquad\qquad\qquad\qquad$ By inversion on $\rho^{conc} \sqsubseteq \rho^{abs}$
$\quad$ $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \mathsf{rel}(\bar{\mathsf{y}})[\sigma]\ \mathsf{t}^{a}$ $\qquad\qquad\qquad\qquad\qquad$ By rule (REL)

Case: $\dfrac{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho \vdash \mathsf{S}[\sigma]\ \mathsf{t}^{c} \qquad \mathcal{B}^{conc}(\mathsf{y}_{test}[\sigma]) = \mathsf{t}^{c} \qquad \mathsf{t}^{c} \neq \mathsf{Unknown}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \mathsf{S}/\mathsf{y}_{test}[\sigma]\ \mathsf{True}}\ \text{(REL\:--TEST\:--T)}$

$\quad$ $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \mathsf{A}\ \mathsf{t}^{a}$ $\qquad\qquad\qquad\qquad\qquad\qquad$ By induction hypothesis
$\quad$ $\mathsf{t}^{c} \sqsubseteq \mathsf{t}^{a}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ By induction hypothesis
$\quad$ By case analysis on t^{c}

$\qquad$ **Case:** $\mathsf{t}^{c} = \mathsf{True}$

$\qquad\qquad$ By case analysis on $\mathcal{B}^{abs}(\ell_{test})$

$\qquad\qquad$ **Case:** $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{True}$
$\qquad\qquad\qquad$ By case analysis on t^{a}
$\qquad\qquad\qquad$ **Case:** $\mathsf{t}^{a} = \mathsf{True}$

$$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{True}$$ By rule (REL-TEST-T)
$$\mathsf{True} \sqsubseteq \mathsf{True}$$ By rule $\sqsubseteq - =$

Case: $t_a = \mathsf{False}$
  Invalid case by $\rho^{conc} \sqsubseteq \rho^{abs}$

Case: $t_a = \mathsf{Unknown}$
  $$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$$ By rule (REL-TEST-U1)
  $$\mathsf{True} \sqsubseteq \mathsf{Unknown}$$ By rule $\sqsubseteq -\mathsf{Unknown}$

Case: $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{False}$
  Invalid case by $\mathcal{B}^{conc} \sqsubseteq_{\mathcal{B}} \mathcal{B}^{abs}$

Case: $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{Unknown}$
  $$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$$ By rule (REL-TEST-U2)

Case: $t^c = \mathsf{False}$

By case analysis on $\mathcal{B}^{abs}(\ell_{test})$

Case: $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{False}$
  By case analysis on t^a
  Case: $t^a = \mathsf{False}$
    $$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{False}$$ By rule (REL-TEST-F)
    $$\mathsf{False} \sqsubseteq \mathsf{False}$$ By rule $\sqsubseteq - =$

  Case: $t^a = \mathsf{True}$
    Invalid case by $\rho^{conc} \sqsubseteq \rho^{abs}$

  Case: $t^a = \mathsf{Unknown}$
    $$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$$ By rule (REL-TEST-U1)
    $$\mathsf{False} \sqsubseteq \mathsf{Unknown}$$ By rule $\sqsubseteq -\mathsf{Unknown}$

Case: $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{True}$
  Invalid case by $\mathcal{B}^{conc} \sqsubseteq_{\mathcal{B}} \mathcal{B}^{abs}$

Case: $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{Unknown}$
  $$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$$ By rule (REL-TEST-U2)

Case: $t^c = \mathsf{Unknown}$

Invalid case by $t^c \neq$ Unknown

Case:
$$\frac{\begin{array}{c} \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho \vdash S[\sigma] \, t_1^c \qquad \mathcal{B}^{conc}(y_{test}[\sigma]) = t_2^c \\ t_1^c \neq \text{Unknown} \qquad t_2^c \neq \text{Unknown} \qquad t_1^c \neq t_2^c \end{array}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \, \text{False}} \text{(REL−TEST−F)}$$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A \, t_1^a$ By induction hypothesis
$t_1^c \sqsubseteq t_1^a$ By induction hypothesis
By case analysis on t_1^c

   Case: $t_1^c = \text{True}$

      $t_2^c = \text{False}$ By $t_1^c \neq t_2^c$ and $t_1^c \neq$ Unknown
      By case analysis on $\mathcal{B}^{abs}(\ell_{test})$

      Case: $\mathcal{B}^{abs}(\ell_{test}) = \text{False}$
         By case analysis on t_1^a
         Case: $t_1^a = \text{True}$
            $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \, \text{False}$ By rule (REL-TEST-F)
            $\text{False} \sqsubseteq \text{False}$ By rule $\sqsubseteq - =$

         Case: $t_1^a = \text{False}$
            Invalid case by $\rho^{conc} \sqsubseteq \rho^{abs}$

         Case: $t_1^a = \text{Unknown}$
            $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \, \text{Unknown}$ By rule (REL-TEST-U1)
            $\text{False} \sqsubseteq \text{Unknown}$ By rule $\sqsubseteq -\text{Unknown}$

      Case: $\mathcal{B}^{abs}(\ell_{test}) = \text{True}$
         Invalid case by $\mathcal{B}^{conc} \sqsubseteq_{\mathcal{B}} \mathcal{B}^{abs}$

      Case: $\mathcal{B}^{abs}(\ell_{test}) = \text{Unknown}$
         $\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \, \text{Unknown}$ By rule (REL-TEST-U2)

   Case: $t_1^c = \text{False}$

      $t_2^c = \text{True}$ By $t_1^c \neq t_2^c$ and $t_1^c \neq$ Unknown
      By case analysis on $\mathcal{B}^{abs}(\ell_{test})$

      Case: $\mathcal{B}^{abs}(\ell_{test}) = \text{True}$

By case analysis on t_1^a

Case: $t_1^a = \mathsf{False}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{False}$ By rule (REL-TEST-F)

$\mathsf{False} \sqsubseteq \mathsf{False}$ By rule $\sqsubseteq - =$

Case: $t_1^a = \mathsf{True}$

Invalid case by $\rho^{conc} \sqsubseteq \rho^{abs}$

Case: $t_1^a = \mathsf{Unknown}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$ By rule (REL-TEST-U1)

$\mathsf{False} \sqsubseteq \mathsf{Unknown}$ By rule $\sqsubseteq -\mathsf{Unknown}$

Case: $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{False}$

Invalid case by $\mathcal{B}^{conc} \sqsubseteq_{\mathcal{B}} \mathcal{B}^{abs}$

Case: $\mathcal{B}^{abs}(\ell_{test}) = \mathsf{Unknown}$

$\mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$ By rule (REL-TEST-U2)

Case: $t_1^c = \mathsf{Unknown}$

Invalid case by $t_1^a \neq \mathsf{Unknown}$

Case: $\dfrac{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S[\sigma] \; \mathsf{Unknown}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \; \mathsf{Unknown}}$ (REL$-$TEST$-$U1)

$\mathcal{B}^{abs}; \rho^{abs} \vdash A \; t^a$ By induction hypothesis

$\mathsf{Unknown} \sqsubseteq t^a$ By induction hypothesis

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$ By rule (REL-TEST-U1)

$\mathsf{Unknown} \sqsubseteq \mathsf{Unknown}$ By rule $\sqsubseteq -\mathsf{Unknown}$

Case: $\dfrac{\mathcal{B}^{conc}(y_{test}[\sigma]) = \mathsf{Unknown} \quad \mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S[\sigma] \; t^a}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash S/y_{test}[\sigma] \; \mathsf{Unknown}}$ (REL$-$TEST$-$U2)

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A \; t^a$ By induction hypothesis

$t^c \sqsubseteq t^a$ By induction hypothesis

$\mathcal{B}^{abs}(\ell_{test}) = \mathsf{Unknown}$ By $\mathcal{B}^{conc} \sqsubseteq_{\mathcal{B}} \mathcal{B}^{abs}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash A/\ell_{test} \; \mathsf{Unknown}$ By rule (REL-TEST-U2)

$\mathsf{Unknown} \sqsubseteq \mathsf{Unknown}$ By rule $\sqsubseteq -\mathsf{Unknown}$

Case: $\dfrac{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A[\sigma] \ \mathsf{Unknown}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A[\sigma] \ \mathsf{Unknown}}$ (¬T–UNKNOWN)

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S \ t_a$	By induction hypothesis
$\mathsf{Unknown} \sqsubseteq t_a$	By induction hypothesis
$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg S \ \mathsf{Unknown}$	By rule (¬S – U)
$\mathsf{Unknown} \sqsubseteq \mathsf{Unknown}$	By rule $\sqsubseteq$ –Unknown

Case: $\dfrac{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A[\sigma]\mathsf{False}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A[\sigma]\mathsf{True}}$ (¬T–T)

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S \ t^a$	By induction hypothesis
$\mathsf{False} \sqsubseteq t^a$	By induction hypothesis

By case analysis on the value of t^a

    Case: $t^a = \mathsf{False}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg S \ \mathsf{True}$	By rule (¬S – T)
$\mathsf{True} \sqsubseteq \mathsf{True}$	By rule $\sqsubseteq - =$

    Case: $t^a = \mathsf{True}$

    Contradiction with $\mathsf{False} \sqsubseteq t^a$

    Case: $t^a = \mathsf{Unknown}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg S \ \mathsf{Unknown}$	By rule (¬S – U)
$\mathsf{True} \sqsubseteq \mathsf{Unknown}$	By rule $\sqsubseteq - =$

Case: $\dfrac{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash A[\sigma]\mathsf{True}}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \neg A[\sigma]\mathsf{False}}$ (¬T–F)

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash S \ t^a$	By induction hypothesis
$\mathsf{True} \sqsubseteq t^a$	By induction hypothesis

By case analysis on the value of t^a

    Case: $t^a = \mathsf{True}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg S \ \mathsf{False}$	By rule (¬S – F)
$\mathsf{False} \sqsubseteq \mathsf{False}$	By rule $\sqsubseteq - =$

Case: $t^a = \mathsf{False}$

Contradiction with $\mathsf{True} \sqsubseteq t^a$

Case: $t^a = \mathsf{Unknown}$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \neg S \; \mathsf{Unknown}$	By rule $(\neg S - u)$
$\mathsf{False} \sqsubseteq \mathsf{Unknown}$	By rule $\sqsubseteq - =$

Case: $\dfrac{}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \mathit{true}\mathsf{True}} \, (\text{TRUE})$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \mathit{true}\mathsf{True}$	By rule (TRUE)
$\mathsf{True} \sqsubseteq \mathsf{True}$	By rule $\sqsubseteq - =$

Case: $\dfrac{}{\mathcal{A}^{conc}; \mathcal{B}^{conc}; \rho^{conc} \vdash \mathit{false}\mathsf{False}} \, (\text{FALSE})$

$\mathcal{A}^{abs}; \mathcal{B}^{abs}; \rho^{abs} \vdash \mathit{false}\mathsf{False}$	By rule (FALSE)
$\mathsf{False} \sqsubseteq \mathsf{False}$	By rule $\sqsubseteq - =$

Remaining cases work as expected for a three value logic.

$\square$

C.3   Consistency

Theorem 3. Consistency

$$\texttt{forall deriv.}$$

$$\mathcal{A} \vdash \rho \text{ consistent}$$
$$f_{alias}(\mathcal{A}, \texttt{instr}) = \mathcal{A}'$$
$$f_{\mathcal{C};\mathcal{A}';\mathcal{B}}(\rho; \texttt{instr}) = \rho', \mathcal{A}''$$

$$\texttt{exists deriv.}$$

$$\mathcal{A}'' \vdash \rho' \text{ consistent}$$

Proof:

$\mathcal{T} = \{\texttt{cons}, \delta, \gamma | \texttt{cons} \in \mathcal{C} \wedge \mathcal{A}'; \mathcal{B}; \rho; \texttt{cons} \vdash \texttt{instr} \hookrightarrow \delta, \gamma\}$

	By inversion on $f_{\mathcal{C};\mathcal{A}';\mathcal{B}}(\rho; \texttt{instr}) = \rho', \mathcal{A}''$
$\mathcal{T}.\texttt{cons} = \mathcal{C}$	By inversion on $f_{\mathcal{C};\mathcal{A}';\mathcal{B}}(\rho; \texttt{instr}) = \rho', \mathcal{A}''$
$\gamma = (\sqcup \mathcal{T}.\gamma)$	By inversion on $f_{\mathcal{C};\mathcal{A}';\mathcal{B}}(\rho; \texttt{instr}) = \rho', \mathcal{A}''$
$\mathcal{A}'' = \mathcal{A}' \Leftarrow \gamma$	By inversion on $f_{\mathcal{C};\mathcal{A}';\mathcal{B}}(\rho; \texttt{instr}) = \rho', \mathcal{A}''$
$\delta = (\sqcup \mathcal{T}.\delta)$	By inversion on $f_{\mathcal{C};\mathcal{A}';\mathcal{B}}(\rho; \texttt{instr}) = \rho', \mathcal{A}''$
$\rho' = \textsf{transfer}(\rho, \mathcal{A}'') \Leftarrow \delta$	By inversion on $f_{\mathcal{C};\mathcal{A}';\mathcal{B}}(\rho; \texttt{instr}) = \rho', \mathcal{A}''$
$\mathcal{A}' \vdash \rho \text{ consistent}$	By Lemma Aliasing Flow preserves Consistency
$\forall \texttt{cons} \in \mathcal{C}$	

$\mathcal{A}'; \mathcal{B}; \rho; \texttt{cons} \vdash \texttt{instr} \hookrightarrow \delta, \gamma$	By construction of $\mathcal{T}$
$\mathcal{A}' \vdash \delta \text{ consistent}$	By Lemma 9(Consistency of a Single Constraint)
$\text{dom}(\gamma) \subseteq \text{dom}(\mathcal{A}'.\mathcal{L})$	By Lemma 9(Consistency of a Single Constraint)

$\mathcal{A}' \vdash \delta \text{ consistent}$	By Lemma 20($\sqcup_\delta$ operator preserves consistency)
$\text{dom}(\gamma) \subseteq \text{dom}(\mathcal{A}'.\mathcal{L})$	By Lemma $\sqcup_\gamma$ preserves domains
$\mathcal{A}'' \vdash \delta \text{ consistent}$	By Lemma $\Leftarrow_A$ preserves consistent
$\mathcal{A}'' \vdash \textsf{transfer}(\rho, \mathcal{A}'') \text{ consistent}$	By Lemma transfer is consistent
$\mathcal{A}'' \vdash \rho' \text{ consistent}$	By Lemma $\Leftarrow$ preserves consistency

$\square$

Lemma 9 (Consistency of a Single Constraint).

$$forall\ deriv.$$
$$\mathcal{A} \vdash \rho \text{ consistent}$$
$$\mathcal{A}; \mathcal{B}; \rho; cons \vdash instr \hookrightarrow \delta, \gamma$$
$$exists\ deriv.$$
$$\mathcal{A} \vdash \delta \text{ consistent}$$
$$dom(\gamma) \subseteq dom(\mathcal{A}.\mathcal{L})$$

Proof:
By case analysis on $\mathcal{A}; \rho; cons \vdash instr \hookrightarrow \delta, \gamma$

Case:
$$\frac{\begin{array}{c} \Gamma_y = FV(op) \cup FV(P_{ctx}) \cup FV(\bar{Q}) \qquad instr : op \mapsto \beta \qquad \mathsf{findLabels}(\mathcal{A}; \Gamma_y; \beta) = \Sigma \\ \Sigma \neq \varnothing \qquad \mathcal{T} = \{\sigma, \delta, \gamma | \sigma \in \Sigma \wedge \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash cons \hookrightarrow \delta, \alpha \wedge \gamma = \alpha[\beta]\} \\ \mathcal{T}.\sigma = \Sigma \qquad \delta' = \sqcup \mathcal{T}.\delta \qquad \gamma' = \sqcup \mathcal{T}.\gamma \end{array}}{\mathcal{A}; \mathcal{B}; \rho; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; P_{rst} \vdash instr \hookrightarrow \delta', \gamma'} \text{(MATCH)}$$

$\forall \sigma \in \Sigma$

$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash cons \hookrightarrow \delta, \alpha$	By construction of $\mathcal{T}$
$\gamma = \alpha[\beta]$	By construction of $\mathcal{T}$
$\mathcal{A} \vdash \delta$ consistent	By Lemma 10(Consistency of Full Binding)
$rng(\beta) \subseteq dom(\mathcal{A}.\mathcal{L})$	By Lemma matching uses valid variables
$dom(\gamma) \subseteq dom(\mathcal{A}.\mathcal{L})$	By substitution

$\mathcal{A} \vdash \delta$ consistent	By Lemma $\sqcup$ preserves consistency
$dom(\gamma') \subseteq dom(\mathcal{A}.\mathcal{L})$	By Lemma $\sqcup$ preserves domain

Case:
$$\frac{cons = op : P_{ctx} \Rightarrow P_{req} \Downarrow \overline{Q}; P_{rst} \qquad instr : op \mapsto \beta \qquad \mathsf{findLabels}(\mathcal{A}; \Gamma_y; \beta) = \varnothing}{\mathcal{A}; \mathcal{B}; \rho; op : P_{ctx} \Rightarrow P_{req} \Downarrow \bar{Q}; \vdash instr \hookrightarrow \mathsf{ignore}(\mathcal{A}), \varnothing} \text{(NO-ALIASES)}$$

$\mathcal{A} \vdash \mathsf{ignore}(\mathcal{A})$ consistent	By Lemma ignore is consistent
$\varnothing \subseteq dom(\mathcal{A}.\mathcal{L})$	By rule $(\subseteq -\varnothing)$

Case:
$$\frac{cons = op : P_{ctx} \Rightarrow P_{req} \Downarrow \overline{Q}; P_{rst} \qquad \neg(instr : op \mapsto \beta)}{\mathcal{A}; \mathcal{B}; \rho; cons \vdash instr \hookrightarrow \mathsf{ignore}(\mathcal{A}), \varnothing} \text{(NO-MATCH)}$$

$\mathcal{A} \vdash$ ignore$(\mathcal{A})$ consistent   By Lemma ignore is consistent
$\varnothing \subseteq \mathrm{dom}(\mathcal{A}.\mathcal{L})$ By rule $(\subseteq -\varnothing)$

$\square$

Lemma 10 (Consistency of Full Binding).

$$forall\ deriv.$$

$$\mathrm{cons} = \mathrm{op} : \mathrm{P_{ctx}} \Rightarrow \mathrm{P_{req}} \Downarrow \overline{\mathrm{Q}}; \mathrm{P_{rst}}$$
$$\mathcal{A} \vdash \sigma\ \mathsf{validFor}\ \mathrm{FV}(\bar{\mathrm{Q}})$$
$$\mathcal{A} \vdash \rho\ \mathsf{consistent}$$
$$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \mathrm{cons} \hookrightarrow \delta, \alpha$$

$$exists\ deriv.$$

$$\mathcal{A} \vdash \delta\ \mathsf{consistent}$$
$$\mathrm{dom}(\alpha) = \mathrm{dom}(\sigma)$$

Proof:

By case analysis on all variants of $\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \mathrm{cons} \hookrightarrow \delta, \alpha$

$$\textbf{Case:}\ \frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash \mathrm{P_{ctx}}[\sigma]\ \mathsf{True} \\ \mathsf{allValidSubs}(\mathcal{A}; \sigma; \mathrm{FV}(\mathrm{cons})) = \Sigma \qquad \exists\ \sigma' \in \Sigma\ .\ \mathcal{A}; \mathcal{B}; \rho \vdash \mathrm{P_{req}}[\sigma']\ \mathsf{True} \\ \mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{\mathrm{Q}}) = \delta \qquad \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha \mathrm{P_{rst}} \hookrightarrow \alpha \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \mathrm{cons} \hookrightarrow \delta, \alpha}\ (\textsc{bound-t-pragmatic})$$

$\mathcal{A} \vdash \delta$ consistent   By Lemma 11(Consistency of Lattice)
$\mathrm{dom}(\alpha) = \mathrm{dom}(\sigma)$ By Lemma 12(Consistency of Restriction)

$$\textbf{Case:}\ \frac{\mathcal{A}; \mathcal{B}; \rho \vdash \mathrm{P_{ctx}}[\sigma]\ \mathsf{False}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \mathrm{cons} \hookrightarrow \perp(\mathcal{A}), \sigma}\ (\textsc{bound-f-pragmatic})$$

$\mathcal{A} \vdash \perp(\mathcal{A})$ consistent   By Lemma $\perp$ consistent
$\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma)$ By equality

$$\textbf{Case:}\ \frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash \mathrm{P_{ctx}}[\sigma]\ \mathsf{Unknown} \\ \mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{\mathrm{Q}}) = \delta \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \mathrm{cons} \overset{\uparrow}{\hookrightarrow}_* \delta, \sigma}\ (\textsc{bound-u-pragmatic})$$

$\mathcal{A} \vdash \delta$ consistent $\hspace{2cm}$ By Lemma 11(Consistency of Lattice)

$\mathcal{A} \vdash^{\uparrow}_{*} \delta$ consistent $\hspace{2cm}$ By Lemma $\overset{\uparrow}{*}$ preserves consistent

$\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma)$ $\hspace{2cm}$ By equality

Case:
$$\frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{ctx}[\sigma] \; \mathsf{True} \\ \mathsf{allValidSubs}(\mathcal{A}; \sigma; FV(cons)) = \Sigma \hspace{1em} \forall \, \sigma' \in \Sigma \, . \, \mathcal{A}; \mathcal{B}; \rho \vdash P_{req}[\sigma'] \; \mathsf{True} \\ \mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \hspace{1em} \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_{\alpha} P_{rst} \hookrightarrow \alpha \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash cons \hookrightarrow \delta, \alpha} (\text{BOUND--T--SOUND})$$

$\mathcal{A} \vdash \delta$ consistent $\hspace{2cm}$ By Lemma 11(Consistency of Lattice)

$\mathrm{dom}(\alpha) = \mathrm{dom}(\sigma)$ $\hspace{2cm}$ By Lemma 12(Consistency of Restriction)

Case:
$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash P_{ctx}[\sigma] \; \mathsf{False}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash cons \hookrightarrow \bot(\mathcal{A}), \sigma} (\text{BOUND--F--SOUND})$$

$\mathcal{A} \vdash \bot(\mathcal{A})$ consistent $\hspace{2cm}$ By Lemma $\bot$ consistent

$\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma)$ $\hspace{2cm}$ By equality

Case:
$$\frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{ctx}[\sigma] \; \mathsf{Unknown} \\ \mathsf{allValidSubs}(\mathcal{A}; \sigma; FV(cons)) = \Sigma \hspace{1em} \forall \, \sigma' \in \Sigma \, . \, \mathcal{A}; \mathcal{B}; \rho \vdash P_{req}[\sigma'] \; \mathsf{True} \\ \mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash cons \hookrightarrow^{\uparrow}_{*} \delta, \sigma} (\text{BOUND--U--SOUND})$$

$\mathcal{A} \vdash \delta$ consistent $\hspace{2cm}$ By Lemma 11(Consistency of Lattice)

$\mathcal{A} \vdash^{\uparrow}_{*} \delta$ consistent $\hspace{2cm}$ By Lemma $\overset{\uparrow}{*}$ preserves consistent

$\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma)$ $\hspace{2cm}$ By equality

Case:
$$\frac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{ctx}[\sigma] \; \mathsf{True} \hspace{1em} \mathsf{allValidSubs}(\mathcal{A}; \sigma; FV(cons)) = \Sigma \\ \exists \, \sigma' \in \Sigma \, . \, \mathcal{A}; \mathcal{B}; \rho \vdash P_{req}[\sigma'] \; \mathsf{True} \vee \mathcal{A}; \mathcal{B}; \rho \vdash P_{req}[\sigma'] \; \mathsf{Unknown} \\ \mathsf{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \hspace{1em} \mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_{\alpha} P_{rst} \hookrightarrow \alpha \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash cons \hookrightarrow \delta, \alpha} (\text{BOUND--T--COMPLETE})$$

$\mathcal{A} \vdash \delta$ consistent $\hspace{2cm}$ By Lemma 11(Consistency of Lattice)

$\mathrm{dom}(\alpha) = \mathrm{dom}(\sigma)$ $\hspace{2cm}$ By Lemma 12(Consistency of Restriction)

Case:
$$\frac{\mathcal{A}; \mathcal{B}; \rho \vdash P_{ctx}[\sigma] \; \mathsf{False}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash cons \hookrightarrow \bot(\mathcal{A}), \sigma} (\text{BOUND--F--COMPLETE})$$

$\mathcal{A} \vdash \perp(\mathcal{A})$ consistent   By Lemma $\perp$ consistent

$\text{dom}(\sigma) = \text{dom}(\sigma)$ By equality

Case: $\dfrac{\begin{array}{c} \mathcal{A}; \mathcal{B}; \rho \vdash P_{\text{ctx}}[\sigma] \; \text{Unknown} \\ \text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta \end{array}}{\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash \text{cons} \overset{\uparrow}{\hookrightarrow}_* \delta, \sigma}$ (BOUND–U–COMPLETE)

$\mathcal{A} \vdash \delta$ consistent   By Lemma 11(Consistency of Lattice)

$\mathcal{A} \overset{\uparrow}{\vdash}_* \delta$ consistent   By Lemma $\overset{\uparrow}{*}$ preserves consistent

$\text{dom}(\sigma) = \text{dom}(\sigma)$ By equality

$\square$

Lemma 11 (Consistency of Lattice).

$forall\ deriv.$

$\mathcal{A} \vdash \sigma\ \text{validFor}\ \text{FV}(Q)$

$\text{lattice}(\mathcal{A}; \mathcal{B}; \sigma; \bar{Q}) = \delta$

$exists\ deriv.$

$\mathcal{A} \vdash \delta$ consistent

Lemma 12 (Consistency of Restriction).

$forall\ deriv.$

$\mathcal{A}; \mathcal{B}; \rho; \sigma; \vdash_\alpha P \rightarrow \alpha$

$exists\ deriv.$

$\text{dom}(\alpha) = \text{dom}(\sigma)$

Lemma 13 (Consistency and precision implies domains are subset).

$\forall\ deriv.$

$< \Gamma_\ell^c; \mathcal{L}^c > \vdash \rho^c$ consistent

$< \Gamma_\ell^a; \mathcal{L}^a > \vdash \rho^a$ consistent

$< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_{\mathcal{A}} < \Gamma_\ell^a; \mathcal{L}^a >$

$\exists\ deriv.$

$\text{dom}(\rho^c) \subseteq \text{dom}(\rho^a)$

Proof:

$\text{dom}(\rho^c) = \{\text{rel}(\bar{\ell}) \mid \bar{\tau} = \mathcal{R}(\text{rel}) \ \wedge \ |\bar{\tau}| = |\bar{\ell}| = n \ \wedge \ \forall_{i=1}^n \ . \ \exists \tau' \ . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell^c(\ell_i)\}$

By inversion on $< \Gamma_\ell^c; \mathcal{L}^c > \vdash \rho^c$ consistent

$\text{dom}(\rho^a) = \{\text{rel}(\bar{\ell}) \mid \bar{\tau} = \mathcal{R}(\text{rel}) \ \wedge \ |\bar{\tau}| = |\bar{\ell}| = n \ \wedge \ \forall_{i=1}^n \ . \ \exists \tau' \ . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell^a(\ell_i)\}$

By inversion on $< \Gamma_\ell^a; \mathcal{L}^a > \vdash \rho^a$ consistent

$\forall \text{rel}(\bar{\ell}) \in \text{dom}(\rho^c) \ . \ \bar{\tau} = \mathcal{R}(\text{rel}) \ \wedge \ |\bar{\tau}| = |\bar{\ell}| = n \ \wedge \ \forall_{i=1}^n \ . \ \exists \tau' \ . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell^c(\ell_i)$

By construction of $\text{dom}(\rho^c)$

$\text{dom}(\Gamma_\ell^a) = \text{dom}(\Gamma_\ell^c)$ By inversion on $< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_A < \Gamma_\ell^a; \mathcal{L}^a >$

$\forall \ \ell : \tau \in \Gamma_\ell^c . \ \tau <: \Gamma_\ell^a(\ell)$ By inversion on $< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_A < \Gamma_\ell^a; \mathcal{L}^a >$

$\forall \ \text{rel}(\bar{\ell}) \in \text{dom}(\rho^c) \ . \ \bar{\tau} = \mathcal{R}(\text{rel}) \ \wedge \ |\bar{\tau}| = |\bar{\ell}| = n \ \wedge \ \forall_{i=1}^n \ . \ \exists \tau' \ . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell^a(\ell_i)$

By $<:$ transitive

$\forall \ \text{rel}(\bar{\ell}) \in \text{dom}(\rho^c) \ . \ \text{rel}(\bar{\ell}) \in \text{dom}(\rho^a)$ By construction of $\text{dom}(\rho^a)$

$\text{dom}(\rho^c) \subseteq \text{dom}(\rho^a)$ By $\subseteq$

$\square$

Lemma 14 (σ valid and ρ consistent gives $R \in \rho$)**.**

$$\text{forall } deriv.$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \sigma \ \text{validFor FV}(\text{rel}(\bar{y}))$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \rho \ \text{consistent}$$
$$\text{exists } deriv.$$
$$\text{rel}(\bar{y})[\sigma] \in \text{dom}(\rho)$$

Proof:

$\text{dom}(\sigma) \supseteq \text{dom}(\Gamma_y)$ By inversion on $< \Gamma_\ell; \mathcal{L} > \vdash \sigma \ \text{validFor FV}(\text{rel}(\bar{y}))$

$\forall \ y : \tau \in \Gamma_y \ . \ \exists \tau' \ . \ \tau' <: \Gamma_\ell(\sigma(y)) \ \wedge \ \tau' <: \tau$ By inversion on $< \Gamma_\ell; \mathcal{L} > \vdash \sigma \ \text{validFor FV}(\text{rel}(\bar{y}))$

$\text{dom}(\rho) = \{\text{rel}(\bar{\ell}) \mid \bar{\tau} = \mathcal{R}(\text{rel}) \ \wedge \ |\bar{\tau}| = |\bar{\ell}| = n \ \wedge \ \forall_{i=1}^n \ . \ \exists \tau' \ . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell(\ell_i)\}$

By inversion on $< \Gamma_\ell; \mathcal{L} > \vdash \rho \ \text{consistent}$

$\bar{y} = \text{dom}(\text{FV}(\text{rel}(\bar{y})))$ By inversion on FV

Let $\bar{\tau} = \mathcal{R}(\text{rel})$

$\bar{\ell} = \bar{y}[\sigma]$ By $\text{dom}(\sigma) \supseteq \text{dom}(\Gamma_y)$

$|\bar{\ell}| = |\bar{y}| = |\bar{\tau}| = n$ By substitution and typing of rel

Let $\Gamma_y = \text{FV}(\text{rel}(\bar{y}))$

$\Gamma_y = y_0 : \tau_0, \ldots, y_n : \tau_n$ By inversion of FV

$\forall_{i=1}^n \ . \ \exists \tau' \ . \ \tau' <: \tau_i \ \wedge \ \tau' <: \Gamma_\ell(\ell_i)$ By $\text{dom}(\sigma) \supseteq \text{dom}(\Gamma_y)$

$\text{rel}(\bar{y})[\sigma] \in \text{dom}(\rho)$ By construction of the domain of ρ

$\square$

Lemma 15. Consistency implies same domain, ρ/ρ

$$\forall\ deriv.$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \rho_1 \text{ consistent}$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \rho_2 \text{ consistent}$$
$$\exists\ deriv.$$
$$\text{dom}(\rho_1) = \text{dom}(\rho_2)$$

Lemma 16. Consistency implies same domain, ρ/δ

$$\forall\ deriv.$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \rho \text{ consistent}$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \delta \text{ consistent}$$
$$\exists\ deriv.$$
$$\text{dom}(\rho) = \text{dom}(\delta)$$

Lemma 17. Consistency implies same domain, δ/δ

$$\forall\ deriv.$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \delta_1 \text{ consistent}$$
$$< \Gamma_\ell; \mathcal{L} > \vdash \delta_2 \text{ consistent}$$
$$\exists\ deriv.$$
$$\text{dom}(\delta_1) = \text{dom}(\delta_2)$$

Lemma 18. consistent and $\sqsubseteq$ causes subset domains on δ

$$\forall\ deriv.$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{conc}$$
$$\mathcal{A}^{conc} \vdash \delta^{conc}$$
$$\mathcal{A}^{abs} \vdash \delta^{abs}$$
$$\exists\ deriv.$$
$$\text{dom}(\delta^{conc}) \subseteq \text{dom}(\delta^{abs})$$

Lemma 19. consistent and $\sqsubseteq$ causes subset domains on ρ

$$\forall\ deriv.$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{conc}$$
$$\mathcal{A}^{conc} \vdash \rho^{conc}$$
$$\mathcal{A}^{abs} \vdash \rho^{abs}$$
$$\exists\ deriv.$$
$$\text{dom}(\rho^{conc}) \subseteq \text{dom}(\rho^{abs})$$

Lemma 20 ($\sqcup_\delta$ operator preserves consistency)**.**

$$\forall\ deriv.$$
$$\mathcal{A} \vdash \delta_l \text{ consistent}$$
$$\mathcal{A} \vdash \delta_r \text{ consistent}$$
$$\exists\ deriv.$$
$$\delta_l \sqcup \delta_r = \delta$$
$$\mathcal{A} \vdash \delta \text{ consistent}$$

Proof: Trivially true. □

TODO this lemma. Working on Consistency theorem.

Lemma 21. $\sqcup_\gamma$ operator preserves $\sqsubseteq$

$$\forall\ deriv.$$
$$\gamma_l^{conc} \sqsubseteq \gamma_l^{abs}$$
$$\gamma_r^{conc} \sqsubseteq \gamma_r^{abs}$$
$$\gamma_l^{conc} \sqcup \gamma_r^{conc} = \gamma^{conc}$$
$$\gamma_l^{abs} \sqcup \gamma_r^{abs} = \gamma^{abs}$$
$$\exists\ deriv.$$
$$\gamma^{conc} \sqsubseteq \gamma^{abs}$$

Proof: Trivially true. □

C.4 Function Lemmas

Lemma 22 (FindLabels returns subsets).

$$forall\ deriv.$$
$$< \Gamma_\ell^c, \mathcal{L}^c > \sqsubseteq_\mathcal{A} < \Gamma_\ell^a, \mathcal{L}^a >$$
$$\mathrm{dom}(\beta) \subseteq \mathrm{dom}(\Gamma_y)$$
$$\mathrm{rng}(\beta) \subseteq \mathrm{dom}(\mathcal{L}^c)$$
$$\mathrm{rng}(\beta) \subseteq \mathrm{dom}(\mathcal{L}^a)$$

$$exists\ deriv.$$
$$\mathsf{findLabels}(< \Gamma_\ell^a, \mathcal{L}^a >, \Gamma_y, \beta) = \Sigma^a$$
$$\mathsf{findLabels}(< \Gamma_\ell^c, \mathcal{L}^c >, \Gamma_y, \beta) = \Sigma^c$$
$$\Sigma^c \subseteq \Sigma^a$$
$$\forall \sigma \in \Sigma^c.\ < \Gamma_\ell^c, \mathcal{L}^c > \vdash \sigma\ \mathsf{validFor}\ \Gamma_y\ \wedge\ \mathrm{dom}(\sigma) = \mathrm{dom}(\Gamma_y)$$
$$\forall \sigma \in \Sigma^a.\ < \Gamma_\ell^a, \mathcal{L}^a > \vdash \sigma\ \mathsf{validFor}\ \Gamma_y\ \wedge\ \mathrm{dom}(\sigma) = \mathrm{dom}(\Gamma_y)$$

Proof:

$\mathsf{findLabels}(< \Gamma_\ell^a, \mathcal{L}^a >, \Gamma_y, \beta) = \Sigma^a$ By applying function $\mathsf{findLabels}$

$\Sigma^a = \{\sigma' \mid \mathrm{dom}(\sigma) = \mathrm{dom}(\beta)\ \wedge\ \sigma = \{y \mapsto \ell \mid \ell \in \mathcal{L}^a(\beta(y))\ \wedge\ \exists \tau'.\ \tau' <: \Gamma_\ell^a(\ell)\ \wedge\ \tau' <: \Gamma_y(y)\}\ \wedge\ \mathsf{allValidSubs}(<$

$\mathsf{findLabels}(< \Gamma_\ell^c, \mathcal{L}^c >, \Gamma_y, \beta) = \Sigma^c$ 'By applying function $\mathsf{findLabels}$

$\Sigma^c = \{\sigma' \mid \mathrm{dom}(\sigma) = \mathrm{dom}(\beta)\ \wedge\ \sigma = \{y \mapsto \ell \mid \ell \in \mathcal{L}^c(\beta(y))\ \wedge\ \exists \tau'.\ \tau' <: \Gamma_\ell^c(\ell)\ \wedge\ \tau' <: \Gamma_y(y)\}\ \wedge\ \mathsf{allValidSubs}(<$

$\forall \sigma' \in \Sigma^c.$

$\quad$ Let $\sigma = \{y \mapsto \ell \mid \ell \in \mathcal{L}^c(\beta(y))\ \wedge\ \exists \tau'.\ \tau' <: \Gamma_\ell^c(\ell)\ \wedge\ \tau' <: \Gamma_y(y)\}\ \wedge\ \mathrm{dom}(\sigma) = \mathrm{dom}(\beta)$ By set construction

$\quad \mathsf{allValidSubs}(< \Gamma_\ell^c; \mathcal{L}^c >; \sigma; \Gamma_y) = \Sigma^{c'}$ By set construction

$\quad \mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\Gamma_y)$ By subsets

$\quad \mathsf{allValidSubs}(< \Gamma_\ell^a; \mathcal{L}^a >; \sigma; \Gamma_y) = \Sigma^{a'}$ By Lemma 23(ValidSubs returns subsets)

$\quad \Sigma^{c'} \subseteq \Sigma^{a'}$ By Lemma 23(ValidSubs returns subsets)

$\quad \sigma' \in \Sigma^{c'}$ By set construction

$\quad \sigma' \in \Sigma^{a'}$ By subsets

$\quad \sigma' \in \Sigma^a$ By set construction

$\Sigma^c \subseteq \Sigma^a$

$\forall \sigma \in \Sigma^c.\ < \Gamma_\ell^c, \mathcal{L}^c > \vdash \sigma\ \mathsf{validFor}\ \Gamma_y\ \wedge\ \mathrm{dom}(\sigma) = \mathrm{dom}(\Gamma_y)$ By Lemma 23(ValidSubs returns subsets)

$\forall \sigma \in \Sigma^a.\ < \Gamma_\ell^a, \mathcal{L}^a > \vdash \sigma\ \mathsf{validFor}\ \Gamma_y\ \wedge\ \mathrm{dom}(\sigma) = \mathrm{dom}(\Gamma_y)$ By Lemma 23(ValidSubs returns subsets)

$\square$

Lemma 23 (ValidSubs returns subsets)**.**

$$forall\ deriv.$$

$$< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_A < \Gamma_\ell^a; \mathcal{L}^a >$$
$$\text{dom}(\sigma) \subseteq \text{dom}(\Gamma_y)$$

$$exists\ deriv.$$

$$\text{allValidSubs}(< \Gamma_\ell^a; \mathcal{L}^a >; \sigma; \Gamma_y) = \Sigma^a$$
$$\text{allValidSubs}(< \Gamma_\ell^c; \mathcal{L}^c >; \sigma; \Gamma_y) = \Sigma^c$$
$$\forall\ \sigma \in \Sigma^a\ .\ < \Gamma_\ell^a; \mathcal{L}^a > \vdash \sigma\ \text{validFor}\ \Gamma_y\ \wedge\ \text{dom}(\sigma) = \text{dom}(\Gamma_y)$$
$$\forall\ \sigma \in \Sigma^c\ .\ < \Gamma_\ell^c; \mathcal{L}^c > \vdash \sigma\ \text{validFor}\ \Gamma_y\ \wedge\ \text{dom}(\sigma) = \text{dom}(\Gamma_y)$$
$$\Sigma^c \subseteq \Sigma^a$$

Proof:

$\text{allValidSubs}(< \Gamma_\ell^a; \mathcal{L}^a >; \sigma; \Gamma_y) = (\Sigma_a^t, \Sigma_a^u)$	By applying function allValidSubs
Let $\Sigma^a = \{\sigma' \mid \sigma' \supseteq \sigma\ \wedge\ \text{dom}(\sigma') = \text{dom}(\Gamma_y)\ \wedge$	
$\quad \forall\ y \mapsto \ell \in \sigma'\ .\ \exists\ \tau'\ .\ \tau' <: \Gamma_\ell^a(\ell)\ \wedge\ \tau' <: \Gamma_y(y)\}$	
$\forall \sigma \in \Sigma^a\ .\ \text{dom}(\sigma) = \text{dom}(\Gamma_y)\ \wedge\ \forall\ y \mapsto \ell \in \sigma\ .\ \exists\ \tau'\ .\ \tau' <: \Gamma_\ell^a(\ell)\ \wedge\ \tau' <: \Gamma_y(y)$	By construction of Σ^a
$\forall \sigma \in \Sigma^a\ .\ < \Gamma_\ell^a; \mathcal{L}^a > \vdash \sigma\ \text{validFor}\ \Gamma_y$	By rule (σ – VALID)
$\text{allValidSubs}(< \Gamma_\ell^c; \mathcal{L}^c >; \sigma; \Gamma_y) = \Sigma^c$	By applying function allValidSubs
Let $\Sigma^c = \{\sigma' \mid \sigma' \supseteq \sigma\ \wedge\ \text{dom}(\sigma') = \text{dom}(\Gamma_y)\ \wedge$	
$\quad \forall\ y \mapsto \ell \in \sigma'\ .\ \exists\ \tau'\ .\ \tau' <: \Gamma_\ell^c(\ell)\ \wedge\ \tau' <: \Gamma_y(y)\}$	
$\forall \sigma \in \Sigma^c\ .\ \text{dom}(\sigma) = \text{dom}(\Gamma_y)\ \wedge\ \forall\ y \mapsto \ell \in \sigma\ .\ \exists\ \tau'\ .\ \tau' <: \Gamma_\ell^c(\ell)\ \wedge\ \tau' <: \Gamma_y(y)$	By construction of Σ^c
$\forall \sigma \in \Sigma^c\ .\ < \Gamma_\ell^c; \mathcal{L}^c > \vdash \sigma\ \text{validFor}\ \Gamma_y$	By rule (σ – VALID)
$\text{dom}(\mathcal{L}^c) = \text{dom}(\mathcal{L}^a)$	By inversion on $< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_A < \Gamma_\ell^a; \mathcal{L}^a >$
$\text{dom}(\Gamma_\ell^c) \subseteq \text{dom}(\Gamma_\ell^a)$	By inversion on $< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_A < \Gamma_\ell^a; \mathcal{L}^a >$
$\forall\ \ell' : \tau' \in \Gamma_\ell^c.\ \tau' <: \Gamma_\ell^a(\ell')$	By inversion on $< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_A < \Gamma_\ell^a; \mathcal{L}^a >$
$\forall\ x' \mapsto \{\ell\} \in \mathcal{L}^c.\ \{\ell\} \subseteq \mathcal{L}^a(x')\ \wedge\ \{\ell\} \neq \varnothing$	By inversion on $< \Gamma_\ell^c; \mathcal{L}^c > \sqsubseteq_A < \Gamma_\ell^a; \mathcal{L}^a >$
$\forall\ \ell \in \text{dom}(\Gamma_\ell^c)\ .\ \Gamma_\ell^c(\ell) <: \Gamma_\ell^a(\ell)$	By rewriting
$\forall\ x \in \text{dom}(\mathcal{L}^c)\ .\ \mathcal{L}^c(x) \subseteq \mathcal{L}^a(x)\ \wedge\ \mathcal{L}^c(x) \neq \varnothing$	By rewriting
$\forall\ \sigma' \in \Sigma^c\ .$	
$\quad \sigma' \supseteq \sigma$	By construction of σ'
$\quad \text{dom}(\sigma') = \text{dom}(\Gamma_y)$	By construction of σ'
$\quad \forall\ (y \mapsto \ell) \in \sigma'\ .$	
$\quad\quad \exists\ \tau'\ .\ \tau' <: \Gamma_\ell^c(\ell)\ \wedge\ \tau' <: \Gamma_y(y)$	By construction of σ'
$\quad\quad \exists\ \tau'\ .\ \tau' <: \Gamma_\ell^a(\ell)\ \wedge\ \tau' <: \Gamma_y(y)$	By $\Gamma_\ell^c(\ell) <: \Gamma_\ell^a(\ell)$

$\forall\,(y \mapsto \ell) \in \sigma'\,.\,\exists \tau'\,.\,\tau' <: \Gamma_\ell^a(\ell)\;\wedge\;\tau' <: \Gamma_y(y)$

$\sigma' \in \Sigma^a$ By construction of Σ^a

$\Sigma^c \subseteq \Sigma^a$ By quantification above

$\square$

Lemma 24 (Restriction less precise than substitution)**.**

$$\textit{forall deriv.}$$
$$\mathcal{A} \vdash \rho \;\text{consistent}$$
$$\textit{exists deriv.}$$
$$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \mapsto \alpha$$
$$\alpha \sqsubseteq \sigma$$

Proof:

$\Sigma = \mathsf{allValidSubs}(\mathcal{A}; \sigma; FV(P))$ By applying function $\mathsf{allValidSubs}$
By case analysis on the property of Σ

Case: $\exists \sigma' \in \Sigma.\mathcal{A}; \mathcal{B}; \rho \vdash P[\sigma']\;t\;\wedge\;t \neq \mathsf{False}$

$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \mapsto \sigma$ By rule (RESTRICT-T-U-SOUND/COMPLETE)
$\sigma \sqsubseteq \sigma$ By rule $\sqsubseteq_\alpha - =$

Case: $\neg\exists \sigma' \in \Sigma.\mathcal{A}; \mathcal{B}; \rho \vdash P[\sigma']\;t\;\wedge\;t \neq \mathsf{False}$

$\forall \sigma' \in \Sigma.\mathcal{A}; \mathcal{B}; \rho \vdash P[\sigma']\;\mathsf{False}$ By rewriting
$\mathcal{A}; \mathcal{B}; \rho; \sigma \vdash_\alpha P \mapsto \bot(\sigma)$ By rule (RESTRICT-F-SOUND/COMPLETE)
$\bot(\sigma) \sqsubseteq \sigma$ By rule $\sqsubseteq_\alpha - \bot$

$\square$

Lemma 25 (Lattice preserves precision)**.**

$$\mathit{forall\ deriv.}$$
$$\mathcal{A}^c \sqsubseteq \mathcal{A}^a$$
$$\mathcal{B}^c \sqsubseteq \mathcal{B}^a$$
$$\mathcal{A}^c \vdash \sigma\ \mathsf{validFor}\ \mathsf{FV}(\bar{Q})$$
$$\mathcal{A}^a \vdash \sigma\ \mathsf{validFor}\ \mathsf{FV}(\bar{Q})$$
$$\mathrm{dom}(\mathit{sigma}) = \mathrm{dom}(\mathsf{FV}(\bar{Q}))$$
$$\mathit{exists\ deriv.}$$
$$\mathsf{lattice}(\mathcal{A}^c; \mathcal{B}^c; \sigma; \bar{Q}) = \delta^c$$
$$\mathsf{lattice}(\mathcal{A}^a; \mathcal{B}^a; \sigma; \bar{Q}) = \delta^a$$
$$\delta^c \sqsubseteq \delta^a$$

Proof:

By induction on the structure of $\bar{Q}$:

Case: $\bar{Q} = Q, \bar{Q}'$

$\mathsf{lattice}(\mathcal{A}^c; \mathcal{B}^c; \sigma; Q) = \delta_1^c$	By induction hypothesis
$\mathsf{lattice}(\mathcal{A}^a; \mathcal{B}^a; \sigma; Q) = \delta_1^a$	By induction hypothesis
$\delta_1^c \sqsubseteq \delta_1^a$	By induction hypothesis
$\mathsf{lattice}(\mathcal{A}^c; \mathcal{B}^c; \sigma; \bar{Q}') = \delta_2^c$	By induction hypothesis
$\mathsf{lattice}(\mathcal{A}^a; \mathcal{B}^a; \sigma; \bar{Q}') = \delta_2^a$	By induction hypothesis
$\delta_2^c \sqsubseteq \delta_2^a$	By induction hypothesis
Let $\delta^c = \delta_1^c \mathbin{\leftharpoondown} \delta_2^c$	
Let $\delta^a = \delta_1^a \mathbin{\leftharpoondown} \delta_1^a$	
$\mathsf{lattice}(\mathcal{A}^c; \mathcal{B}^c; \sigma; Q, \bar{Q}') = \delta^c$	By rule (LATTICE-LIST)
$\mathsf{lattice}(\mathcal{A}^a; \mathcal{B}^a; \sigma; Q, \bar{Q}') = \delta^a$	By rule (LATTICE-LIST)
$\delta^c \sqsubseteq \delta^a$	By Lemma $\leftharpoondown$ preserves $\sqsubseteq$

Case: $\bar{Q} = \varnothing$

$\mathsf{lattice}(\mathcal{A}^c; \mathcal{B}^c; \sigma; \varnothing) = \mathsf{ignore}(\mathcal{A}^c)$	By rule (LATTICE $- \varnothing$)
$\mathsf{lattice}(\mathcal{A}^a; \mathcal{B}^a; \sigma; \varnothing) = \mathsf{ignore}(\mathcal{A}^a)$	By rule (LATTICE $- \varnothing$)
$\mathsf{ignore}(\mathcal{A}^c) \sqsubseteq \mathsf{ignore}(\mathcal{A}^a)$	By Lemma ignore preserves $\sqsubseteq$

Case: $\bar{Q} = Q$

$\Sigma^c = \text{allValidSubs}(\mathcal{A}^c; \sigma; FV(Q))$	By applying function allValidSubs
$\Sigma^a = \text{allValidSubs}(\mathcal{A}^c; \sigma; FV(Q))$	By applying function allValidSubs
$\Sigma^c \subseteq \Sigma^a$	By Lemma 23(ValidSubs returns subsets)

Let $\delta^{c'} = \delta' = \{R \mapsto E \mid \sigma' \in \Sigma^c \wedge value(\mathcal{B}; Q[\sigma']) = R \mapsto E\}$ Let $\delta^{a'} = \delta' = \{R \mapsto E \mid \sigma' \in \Sigma^a \wedge value(\mathcal{B}; Q$
$\forall \sigma' \in \Sigma^c$.

$\mathcal{A}^c \vdash \sigma'$ validFor $FV(Q)$	By Lemma 23(ValidSubs returns subsets)
$\mathcal{A}^a \vdash \sigma'$ validFor $FV(Q)$	By Lemma 23(ValidSubs returns subsets)
$value(\mathcal{B}^c, Q[\sigma']) = R \mapsto E^c$	By Lemma 26(Lattice value preserves precision)
$value(\mathcal{B}^a, Q[\sigma']) = R \mapsto E^a$	By Lemma 26(Lattice value preserves precision)
$E^c \sqsubseteq E^a$	By Lemma 26(Lattice value preserves precision)

$\forall R \mapsto E^c \in \delta^{c'} . E^c \sqsubseteq \delta^{a'}(R)$	By quantification
$\delta^{c'} \sqsubseteq \delta^{a'}$	By rule $(\sqsubseteq_\delta)$
Let $\delta^c = \text{ignore}(\mathcal{A}^c) \leftharpoondown \delta^{c'}$	
Let $\delta^a = \text{ignore}(\mathcal{A}^a) \leftharpoondown \delta^{a'}$	
$\text{lattice}(\mathcal{A}^c; \mathcal{B}^c; \sigma; Q) = \delta^c$	By rule (LATTICE-Q)
$\text{lattice}(\mathcal{A}^a; \mathcal{B}^a; \sigma; Q) = \delta^a$	By rule (LATTICE-Q)
$\text{ignore}(\mathcal{A}^c) \sqsubseteq \text{ignore}(\mathcal{A}^a)$	By Lemma ignore preserves $\sqsubseteq$
$\delta^c \sqsubseteq \delta^a$	By Lemma $\leftharpoondown$ preserves $\sqsubseteq$

$\square$

Lemma 26 (Lattice value preserves precision)**.**

$$\text{forall deriv.}$$
$$\mathcal{A}^c \sqsubseteq \mathcal{A}^a$$
$$\mathcal{B}^c \sqsubseteq \mathcal{B}^a$$
$$\mathcal{A}^c \vdash \sigma \text{ validFor } FV(\bar{Q})$$
$$\mathcal{A}^a \vdash \sigma \text{ validFor } FV(\bar{Q})$$
$$\text{exists deriv.}$$
$$value(\mathcal{B}^c, Q[\sigma]) = R \mapsto E^c$$
$$value(\mathcal{B}^a, Q[\sigma]) = R \mapsto E^a$$
$$E^c \sqsubseteq E^a$$

Proof:

By induction on the structure of Q:

Case: $Q = S$

  $R = S[\sigma]$
  $value(\mathcal{B}^c, R) = R \mapsto \texttt{true}$ By definition of value
  $value(\mathcal{B}^a, R) = R \mapsto \texttt{true}$ By definition of value
  True $\sqsubseteq$ True   By rule $\sqsubseteq - =$

Case: $Q = \neg S$

  $R = A[\sigma]$
  $value(\mathcal{B}^c, \neg R) = R \mapsto \texttt{false}$ By definition of value
  $value(\mathcal{B}^a, \neg R) = R \mapsto \texttt{false}$ By definition of value
  False $\sqsubseteq$ False   By rule $\sqsubseteq - =$

Case: $Q = S/y$

  $R = S[\sigma]$
  $\ell = \sigma(y)$ $value(\mathcal{B}^c, R/\ell) = R \mapsto \mathcal{B}^c(\ell)$ By definition of value
  $value(\mathcal{B}^a, R/\ell) = R \mapsto \mathcal{B}^a(\ell)$ By definition of value
  $\mathcal{B}^c(\ell) \sqsubseteq \mathcal{B}^a(\ell)$ By inversion on $\mathcal{B}^c \sqsubseteq \mathcal{B}^a$

Case: $Q = \neg S/y$

  $R = S[\sigma]$
  $\ell = \sigma(y)$ $value(\mathcal{B}^c, \neg R/\ell) = R \mapsto \neg\mathcal{B}^c(\ell)$ By definition of value
  $value(\mathcal{B}^a, \neg R/\ell) = R \mapsto \neg\mathcal{B}^a(\ell)$ By definition of value
  $\mathcal{B}^c(\ell) \sqsubseteq \mathcal{B}^a(\ell)$ By inversion on $\mathcal{B}^c \sqsubseteq \mathcal{B}^a$
  $\neg\mathcal{B}^c(\ell) \sqsubseteq \neg\mathcal{B}^a(\ell)$ By Lemma $\neg$ preserves $\sqsubseteq$

                         $\square$

Lemma 27. ignore preserves $\sqsubseteq$

$$\forall \, deriv.$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}abs$$
$$\mathsf{ignore}(\mathcal{A}^{conc}) = \delta conc$$
$$\mathsf{ignore}(\mathcal{A}^{abs}) = \delta abs$$
$$\exists \, deriv.$$
$$\alpha^{conc} \sqsubseteq \alpha abs$$

Lemma 28. findLabels and $\sqsubseteq$ produces subsets

$$\forall \; deriv.$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$
$$\Sigma^{abs} = \text{findLabels}(\mathcal{A}^{abs}; \Gamma_{y_{[}}]; \beta)$$
$$\Sigma^{conc} = \text{findLabels}(\mathcal{A}^{conc}; \Gamma_{y_{[}}]; \beta)$$
$$\exists \; deriv.$$
$$\Sigma^{conc} \subseteq \Sigma^{abs}$$

C.5 Operator Lemmas

All proofs in this section are omitted as they are trivially reproducible from the rules of the operators.

Lemma 29. $\sqcup_\delta$ operator preserves $\sqsubseteq$

$$\forall \text{ deriv.}$$
$$\delta_l^{conc} \sqsubseteq \delta_l^{abs}$$
$$\delta_r^{conc} \sqsubseteq \delta_r^{abs}$$
$$\delta_l^{conc} \sqcup \delta_r^{conc} = \delta^{conc}$$
$$\delta_l^{abs} \sqcup \delta_r^{abs} = \delta^{abs}$$
$$\exists \text{ deriv.}$$
$$\delta^{conc} \sqsubseteq \delta^{abs}$$

Lemma 30 (eqjoin delta operator preserves alap).

$$\forall \text{ deriv.}$$
$$\delta_l^{conc} \sqsubseteq \delta_l^{abs}$$
$$\delta_r^{conc} \sqsubseteq \delta_r^{abs}$$
$$\delta_l^{conc} \boxminus \delta_r^{conc} = \delta^{conc}$$
$$\delta_l^{abs} \boxminus \delta_r^{abs} = \delta^{abs}$$
$$\exists \text{ deriv.}$$
$$\delta^{conc} \sqsubseteq \delta^{abs}$$

Note: The proof for this is a tedious case-by-case proof, and I used the Agda lemma prover to verify that all cases were covered.

Lemma 31. $\overset{\uparrow}{*}_\delta$ operator preserves $\sqsubseteq$

$$\forall \text{ deriv.}$$
$$\delta^{conc} \sqsubseteq \delta^{abs}$$
$$\exists \text{ deriv.}$$
$$\overset{\uparrow}{*} \delta^{conc} \sqsubseteq \overset{\uparrow}{*} \delta^{abs}$$

Lemma 32. $\leftmapsto_\delta$ operator preserves $\sqsubseteq$

$$\forall\ deriv.$$
$$\delta_l^{conc} \sqsubseteq \delta_l^{abs}$$
$$\delta_r^{conc} \sqsubseteq \delta_r^{abs}$$
$$\delta_l^{conc} \leftmapsto \delta_r^{conc} = \delta^{conc}$$
$$\delta_l^{abs} \leftmapsto \delta_r^{abs} = \delta^{abs}$$
$$\exists\ deriv.$$
$$\delta^{conc} \sqsubseteq \delta^{abs}$$

Lemma 33 ($\sqcup_\gamma$ operator preserves $\sqsubseteq$)**.**

$$\forall\ deriv.$$
$$\gamma_l^{conc} \sqsubseteq \gamma_l^{abs}$$
$$\gamma_r^{conc} \sqsubseteq \gamma_r^{abs}$$
$$\gamma_l^{conc} \sqcup \gamma_r^{conc} = \gamma^{conc}$$
$$\gamma_l^{abs} \sqcup \gamma_r^{abs} = \gamma^{abs}$$
$$\exists\ deriv.$$
$$\gamma^{conc} \sqsubseteq \gamma^{abs}$$

Lemma 34 (Leftarrow AliasA preserves alap)**.**

$$\forall\ deriv.$$
$$\mathcal{A}^{conc} \sqsubseteq \mathcal{A}^{abs}$$
$$\gamma^{conc} \sqsubseteq \gamma^{abs}$$
$$\mathcal{A}^{conc} \Leftarrow \gamma^{conc} = \mathcal{A}^{conc'}$$
$$\mathcal{A}^{abs} \Leftarrow \gamma^{abs} = \mathcal{A}^{abs'}$$
$$\exists\ deriv.$$
$$\mathcal{A}^{conc'} \sqsubseteq \mathcal{A}^{abs'}$$

Lemma 35. $\sqcup_\delta$ less precise than operands

$$\forall\ deriv.$$
$$d1 : \delta = \delta_l \sqcup \delta_r$$
$$\exists\ deriv.$$
$$d2 : \delta_l \sqsubseteq \delta$$
$$d3 : \delta_r \sqsubseteq \delta$$

Lemma 36. $\sqcup_\gamma$ less precise than operands

$$\forall\ deriv.$$
$$d1 : \gamma = \gamma_l \sqcup \gamma_r$$
$$\exists\ deriv.$$
$$d2 : \gamma_l \sqsubseteq \gamma$$
$$d3 : \gamma_r \sqsubseteq \gamma$$

Lemma 37. $\overset{\uparrow}{*}_\delta$ less precise than operand

$$\forall\ deriv.$$
$$\delta' =\overset{\uparrow}{*} \delta$$
$$\exists\ deriv.$$
$$\delta \sqsubseteq\overset{\uparrow}{*} \delta'$$

Lemma 38 ($\Leftarrow_\rho$ preserves $\sqsubseteq$)**.**

$$\forall\ deriv.$$
$$d1 : \rho^{conc} \sqsubseteq \rho^{abs}$$
$$d2 : \delta^{conc} \sqsubseteq \delta^{abs}$$
$$d3 : \rho^{conc} \Leftarrow \delta^{conc} = \rho^{conc'}$$
$$d4 : \rho^{abs} \Leftarrow \delta^{abs} = \rho^{abs'}$$
$$\exists\ deriv.$$
$$d5 : \rho^{conc'} \sqsubseteq \rho^{abs'}$$

Lemma 39. Substitution preserves $\sqsubseteq_\alpha$

$$\forall\ deriv.$$
$$d1 : \alpha^{conc} \sqsubseteq \alpha abs$$
$$d2 : \alpha^{conc}[\beta] = \alpha conc'$$
$$d3 : \alpha^{abs}[\beta] = \alpha abs'$$
$$\exists\ deriv.$$
$$d4 : \alpha^{conc'} \sqsubseteq \alpha abs'$$

Bibliography

[1] Archive of the example projects used in this dissertation and studied in table 6.4, . URL `http://www.cs.cmu.edu/~cchristo/docs/fusionexamples.zip`.

[2] Archive of the asp.net and spring forum threads discussed in this dissertation, . URL `http://www.cs.cmu.edu/~cchristo/docs/forumposts.zip`.

[3]

[4] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/103135.103138.

[5] ABLE research group. AcmeStudio. URL `http://www.cs.cmu.edu/~acme/AcmeStudio/`.

[6] Agitar. Agitar TestOne. URL `http://www.agitar.com`.

[7] Jong-hoon An, Avik Chaudhuri, and Jeffrey S. Foster. Static typing for ruby on rails. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 590–594, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: http://dx.doi.org/10.1109/ASE.2009.80. URL `http://dx.doi.org/10.1109/ASE.2009.80`.

[8] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 57–74, 2006. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167479.

[9] Apache. Hadoop. URL `http://hadoop.apache.org/`.

[10] Stephanie Balzer, Thomas Gross, and Patrick Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *Proc. of the European Conference on Object Oriented Programming*, 2007.

[11] "bashaasnoot" and "vivek_iit". Dynamic user controls, 2006. URL `http://forums.asp.net/thread/1419194.aspx`.

[12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition edition, 2003.

[13] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 1–6, 1989.

[14] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010. ISSN 0001-0782.

[15] Kevin Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, Carnegie Mellon University, 2009.

[16] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2007.

[17] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *Proc. of the European Conference on Object Oriented Programming*, 2005.

[18] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006.

[19] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proc. of the European Conference on Object Oriented Programming*, 2007.

[20] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proc. of the symposium on Foundations of Software Engineering*, 2008.

[21] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 1075–1075. Springer Berlin / Heidelberg, 2003. URL `http://dx.doi.org/10.1007/3-540-44898-5_4`.

[22] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: http://doi.acm.org/10.1145/1640089.1640108. URL `http://doi.acm.org/10.1145/1640089.1640108`.

[23] Richard Buchanan. Wicked problems in design thinking. *Design Issues*, 8(2):pp. 5–21, 1992. ISSN 07479360. URL `http://www.jstor.org/stable/1511637`.

[24] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991. ISBN 0-89791-428-7. doi: http://doi.acm.org/10.1145/113445.113469.

[25] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297069. URL `http://doi.acm.org/10.1145/1297027.1297069`.

[26] Trisha Quan Ciera Jaspan and Jonathan Aldrich. Error reporting logic. In *Proceedings of the Conference on Automated Software Engineering*, 2008.

[27] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: http://doi.acm.org/10.1145/286936.286947. URL `http://doi.acm.org/10.1145/286936.286947`.

[28] Coverity. Coverity Static Analysis. URL `http://www.coverity.com/products/static-analysis.html`.

[29] "CuriousHARD" and "Marten Deinum". How to handle this situation using awfc, 2007. URL `http://forum.springsource.org/showthread.php?39480`.

[30] W. Lowe D. Heuzeroth, S. Mandel. Generating design pattern detectors from pattern specifications. In *18th IEEE International Conference on Automated Software Engineering*, 2003.

[31] David Heinemeier Hansson. Ruby on Rails. URL `http://rubyonrails.org/`.

[32] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proc. of the European Conference on Object Oriented Programming*, 2004.

[33] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proc. of the International Conference on Software Engineering*, 1996.

[34] "dr_pompeii" and "Marten Deinum". problem:migration spring mvc to swf, 2006. URL `http://forum.springsource.org/showthread.php?36109`.

[35] Facebook. Facebook API. URL `http://developers.facebook.com/`.

[36] George Fairbanks, David Garlan, and William Scherlis. Design fragments make using frameworks easier. In *OOPSLA*, pages 762–763, 2006. ISBN 1-59593-491-X. doi: http://doi.acm.org/10.1145/1176617.1176713.

[37] FindBugs. FindBugs Project, . URL `http://findbugs.sourceforge.net`.

[38] FindBugs. FindBugs Bug Descriptions, . URL `http://findbugs.sourceforge.net/bugDescriptions.html`.

[39] Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, 2006. ISBN 1-59593-027-2. doi: http://doi.acm.org/10.1145/1111037.1111059.

[40] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2002.

[41] Fortify Software. Fortify SCA. URL `http://www.fortifysoftware.com`.

[42] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html`, 2004.

[43] Gary Froehlich, H. James Hoover, Ling Liu, and Paul Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the 19th International Conference on Software engineering*, 1997.

[44] P. van Winsen G. Florijn, M. Meijers. Tool support for object-oriented patterns. In *ECOOP*, LNCS. Springer, 1997.

[45] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: http://doi.acm.org/10.1145/1806799.1806806. URL `http://doi.acm.org/10.1145/1806799.1806806`.

[46] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: why reuse is so hard. *Software, IEEE*, 12(6):17 –26, November 1995. ISSN 0740-7459. doi: 10.1109/52.469757.

[47] Jack Greenfield and Keith Short. *Software Factories: Assembling applications with patterns, models, frameworks, and tools*. Wiley Publishing, Inc., 2004.

[48] MIT Program Analysis Group. Daikon project page. URL `http://groups.csail.mit.edu/pag/daikon/`.

[49] "gurnard", "Colin Yates", and "kfranklin". posting form to self, 2006. URL `http://forum.springsource.org/showthread.php?28603`.

[50] David Heinemeier Hansson. Creating a weblog in 15 minutes with rails 2. URL `http://media.rubyonrails.org/video/rails_blog_2.mov`.

[51] Rob Harrop and Jan Machacek. *Pro Spring*. A-Press, 2005.

[52] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, 1990.

[53] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, pages 122–138, London, UK, 1998. Springer-Verlag. ISBN 3-540-64302-8. URL `http://portal.acm.org/citation.cfm?id=645392.651876`.

[54] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: http://doi.acm.org/10.1145/1328438.1328472. URL `http://doi.acm.org/10.1145/1328438.1328472`.

[55] Daqing Hou and H. James Hoover. Using SCL to specify and check design intent in source code. *IEEE Trans. Softw. Eng.*, 32(6), 2006.

[56] "ilpata", "Rossen Stoyanchev", and "Marten Deinum". problem with viewresolver order attribute, 2006. URL `http://forum.springsource.org/showthread.php?36891`.

[57] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/505145.505149.

[58] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *ECOOP*, 2009.

[59] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA*, 1992.

[60] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10), 1997.

[61] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, and Rick Evans. The spring framework - reference documentation, version 2.0.8. URL `http://static.springsource.org/spring/docs/2.0.8/reference/`.

[62] Joshua Bloch. How to design a good api and why it matters. URL `http://www.infoq.com/presentations/effective-api-design`.

[63] Juergen Hoeller. Jpetstore sample application, version 2.0.8. URL `http://sourceforge.net/projects/springframework/files/springframework-2/2.0.8/spring-framework-2.0.8-with-dependencies.zip/download`. Found in the zip file under samples/JPetStore.

[64] Klocwork. Klocwork K7. URL `http://www.klocwork.com`.

[65] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular Pluggable Analyses for Data Structure Consistency. *IEEE Trans. Softw. Eng.*, 32(12), 12 2006.

[66] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3), 2006.

[67] Choonghwan Lee, Feng Chen, and Grigore Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 591–600, 2011.

[68] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: http://doi.acm.org/10.1145/1926385.1926389. URL `http://doi.acm.org/10.1145/1926385.1926389`.

[69] David Lo and Shahar Maoz. Mining hierarchical scenario-based specifications. In *Proceedings of the Conference on Automated Software Engineering*, 2009.

[70] Sure Logic. JSure for Concurrency. URL `http://www.surelogic.com/concurrency-tools.html`.

[71] David Mandelin, Lin Xu, Rastislav Bod, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2005.

[72] Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231, 2006. ISBN 1-59593-027-2. doi: http://doi.acm.org/10.1145/1111037.1111057.

[73] Microsoft. ASP.NET. URL `http://www.asp.net/`.

[74] Microsoft. Asp.net page lifecycle overview, . URL `http://msdn2.microsoft.com/en-us/library/ms178472.aspx`.

[75] Microsoft. C# version 3.0 specification, . URL `http://msdn.microsoft.com/en-us/library/ms364047(v=vs.80).aspx`.

[76] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, January 2005. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1044834.1044835. URL `http://doi.acm.org/10.1145/1044834.1044835`.

[77] Nomair A. Naeem. Personal communication at OOPSLA, 2008.

[78] Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2008.

[79] Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates in the presence of inter-object references. In *Proc. of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[80] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[81] "nyker", "dr_pompeii", and "Marten Deinum". transition on-exception not working, 2006. URL `http://forum.springsource.org/showthread.php?43182`.

[82] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proc. of the symposium on Principles of Programming Languages*, 2008.

[83] PLAID Research Group. The crystal static analysis framework. URL `http://code.google.com/p/crystalsaf`.

[84] "pompiuses" and "Marten Deinum". Applicationobjectsupport always returns null for applicationcontext, 2006. URL `http://forum.springsource.org/showthread.php?32429`.

[85] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the Conference on Automated Software Engineering*, 2009.

[86] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking api protocol conformance with mined multi-object specifications. Under Submission, 2011.

[87] "raydawg" and "jeremyg484". Struts and webflow confusion, 2006. URL `http://forum.springsource.org/showthread.php?38940`.

[88] Microsoft Research. Spec#. URL `http://research.microsoft.com/en-us/projects/specsharp/`.

[89] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *OOPSLA*, pages 117–133, 1998. ISBN 1-58113-005-8. doi: http://doi.acm.org/10.1145/286936.286951.

[90] Thomas Risberg, Rick Evans, and Portia Tung. Spring mvc step-by-step. URL `http://static.springsource.org/docs/Spring-MVC-step-by-step/`.

[91] "roseability", "DWesthead", "bitmask", and "mokeefe". Codebehind with master pages, 2006. URL `http://forums.asp.net/thread/1422132.aspx`.

[92] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/514188.514190.

[93] "senthilnathan74" and "Marten Deinum". Question on having same view..., 2006. URL `http://forum.springsource.org/showthread.php?39209`.

[94] "sharkman" and "Fredrik N". Binding to a DropDownList membership roles, 2006. URL `http://forums.asp.net/thread/1415249.aspx`.

[95] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: http://doi.acm.org/10.1145/1926385.1926390. URL `http://doi.acm.org/10.1145/1926385.1926390`.

[96] "sokol", "Keith Donald", and "robh". Binding object using form taglib in swf?, 2006. URL `http://forum.springsource.org/showthread.php?26787`.

[97] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[98] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-983768-X.

[99] SpringSource. Phonebook sample application, version 2.0.8, . URL `http://sourceforge.net/projects/springframework/files/springframework-2/2.0.8/spring-framework-2.0.8-with-dependencies.zip/download`.

[100] SpringSource. SpringIDE, . URL `http://www.springsource.org/springide/release-20`.

[101] a division of VMWare SpringSource. The standard for enterprise java development. URL `http://www.springsource.com/developer/spring`.

[102] "strangewill", "vivek_iit", "TonyAlicea", and "sreejukg". Control disappearing, 2006. URL `http://forums.asp.net/thread/1419089.aspx`.

[103] Sunacle. Iterator JavaDoc, . URL `http://download.oracle.com/javase/6/docs/api/java/util/Iterator.html`.

[104] Sunacle. The swing gui framework, version 6, . URL `http://download.oracle.com/javase/6/docs/api/javax/swing/package-summary.html`.

[105] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and ric Tanter. First-class state change in plaid. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

[106] The Eclipse Foundation. The Eclipse Project. URL `http://www.eclipse.org/`.

[107] The JUnit Project. JUnit. URL `http://www.junit.org/`.

[108] The Krell Institute. Open|SpeedShop. URL `http://www.openspeedshop.org/`.

[109] The OpenMPI Project. OpenMPI. URL `http://www.open-mpi.org/`.

[110] John Vlissides. Protection, Part 1: The Hollywood Principle. *C++ Report*, February 1996.

[111] VMWare. Spring. URL `http://www.springsource.com/`.

[112] Robert J. Walker and Kevin Viggers. Implementing Protocols via Declarative Event Patterns. In *Proc. of the symposium on Foundations of Software Engineering*, 2004.

[113] Craig Walls. *Spring in Action*. Manning, 2008.

[114] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, 2004. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996859.